

Intelligence Artificielle
Résolution de problèmes
par l'Homme et la machine

Jean-Louis Laurière

Table des matières

1	Intelligence artificielle	1
1.1	Statut de l'informatique.....	1
1.2	L'intelligence artificielle	2
1.3	Définition et domaines de l'intelligence artificielle	2
1.3.1	Définition	2
1.3.2	Domaines	3
1.4	Historique	5
1.5	Résultats	6
2	Représenter un problème	9
2.1	Introduction.....	9
2.2	Le langage naturel	10
2.3	Poser un problème.....	10
2.4	Les énoncés fermés	10
2.5	La démarche générale en résolution de problème	12
2.6	Un exemple complet	15
2.7	Pour résoudre un problème vous devez successivement.....	18
2.8	Petite histoire des mathématiques et de leur enseignement	19
2.8.1	Les notations au cours des âges	19
2.8.2	L'enseignement	24
2.9	Les représentations	25
2.9.1	Les notations mathématiques	25
2.9.2	Les notations linéaires	26
2.9.3	Notations non linéaires	28
2.9.4	Représentation en machine des expressions	29
2.10	Les modélisation graphiques en Intelligence Artificielle	32
2.10.1	Représentation des connaissances en démonstration automatique :	32
2.10.2	Représentations graphiques en résolution automatique de pro- blèmes :	37
2.10.3	Représentations graphiques en compréhension du langage naturel :	38
2.11	Savoir changer de représentations	40
2.12	Le langage LISP	42
2.12.1	Eléments de base du langage LISP	42
2.12.2	Représentation et évaluation	44
2.12.3	Autres fonctions	47
2.12.4	Lambda définition et propriétés	50
2.12.5	Constructions nouvelles	51
2.13	Les graphes	54
3	Les Systèmes Formels	57

3.1	Introduction	57
3.2	Définition d'un système formel	58
3.2.1	Quelques définitions supplémentaires	59
3.2.2	Règle du jeu valable dans tout SF : règle de substitution	60
3.2.3	Exemples de S.F.	60
3.3	Intérêt des systèmes formels : Décidabilité et interprétation	61
3.3.1	Décidabilité	61
3.3.2	Interprétation	64
3.3.3	Démonstration et valeurs de vérité :	64
3.3.4	Exemples de démonstrations	66
3.4	La logique des propositions : (LP)	69
3.4.1	Interprétation de la logique des propositions.....	72
3.4.2	Décidabilité de la logique des propositions	73
3.5	Le calcul des prédictats du premier ordre	74
3.5.1	L'arithmétique formelle	76
3.5.2	Deuxième théorème de Gödel (1931) : Incomplétude de l'arithmétique.....	78
3.5.3	Mécanisme de la preuve de Gödel.....	80
3.6	Les théorèmes de limitation dans les systèmes formels	84
3.7	L'algorithme d'unification	86
3.7.1	Le problème	86
3.7.2	Principe de l'algorithme	88
3.7.3	La procédure de substitution	90
3.7.4	Finitude de l'algorithme d'unification	93
3.7.5	« Variables » non substituables : indéterminées.....	95
3.7.6	Mise en oeuvre de l'unification	97
3.8	Utilisation de l'unification	100
3.8.1	Problèmes d'analogie.....	100
3.9	Utilisation de l'unification	101
3.9.1	Problèmes d'analogie.....	102
3.9.2	Commentaires sur ce type de problèmes de tests géométriques ..	105
3.9.3	Utilisation de l'algorithme d'unification en démonstration automatique	108
3.9.4	La méthode de Siklóssy et Marinov	108
3.9.5	Résolution d'équations trigonométriques : PRET (M. Grandbastfen (1974))......	111
3.9.6	Résolution d'exercices d'arithmétique : PARI (D. Bourgoin (1978))	116
3.10	Le programme de J. PITRAT en logique des propositions	120
3.10.1	Hiérarchie des métathéories	120
3.10.2	Les métathéories du programme :	121
3.10.3	Choix des essais :	123
3.11	Le Principe de Résolution et le Langage PROLOG	124
3.11.1	Le Principe de Résolution	124
3.11.2	Théorème d'Herbrand	126
3.11.3	Organisation pratique des démonstrations par le principe d'Herbrand	128
3.11.4	Exemples de démonstrations suivant le principe d'Herbrand : ..	130
3.11.5	Analyse de ces démonstrations.	133
3.11.6	Stratégies	133
3.11.7	Intérêts et limites de la Résolution.....	134

3.11.8 Le langage PROLOG et les applications de la méthode de Résolution	135
4 Les méthodes classiques en résolution de problèmes	139
4.1 Exemples de bons algorithmes	139
4.2 Liste des problèmes bien résolus (algorithmes polynômiaux)	157
4.3 Le classement des problèmes selon leur complexité	160
4.3.1 Trois classes de problèmes :	161
4.4 La classe des problèmes NP : non déterministes polynômiaux	163
4.4.1 Deux familles de machines	163
4.5 Liste de problèmes dans <i>NP</i>	166
4.6 Etude des problèmes de <i>NP</i> par classes d'équivalence	168
4.7 Le théorème fondamental (Cook (1971))	168
4.8 La classe des <i>NP</i> -complets	171
4.9 Quelques démonstrations d'équivalences entre problèmes	173
5 Les méthodes par propagation et énumération	179
5.1 Les méthodes de résolution par énumération	179
5.1.1 La classe des problèmes combinatoires	179
5.2 Les méthodes de gradient	180
5.3 La programmation linéaire	182
5.3.1 Formalisation de l'algorithme du simplexe	185
5.4 Méthodes de gradient en théorie des graphes	185
5.4.1 Démonstration de l'algorithme de Kruskal	186
5.5 Recherche heuristique	187
5.6 L'algorithme <i>A</i> *	189
5.7 L'énumération implicite par propagation de contraintes	194
5.7.1 Huit dames, première version	197
5.7.2 Implications liées à un choix	198
5.7.3 Huit dames, version 2	199
5.8 Choix de l'ordre des choix	201
5.9 Vers la programmation dynamique	205
5.9.1 Théorie de la programmation dynamique	206
5.9.2 Intérêt de la programmation dynamique	207
5.10 Coloration optimale des sommets d'un graphe	209
5.10.1 Organisation d'un colloque	210
5.11 Algorithme de coloration optimale d'un graphe	211
5.11.1 Première étape : recherche d'une solution	212
5.11.2 Deuxième étape : recherche de l'optimalité	215
5.12 Le problème du voyageur de commerce	219
5.12.1 Première étape : recherche d'une solution	220
5.12.2 Seconde étape : recherche de l'optimum	222
5.13 Un programme général de résolution de problèmes	227
5.13.1 La démarche générale de GPS : les sous-problèmes et les plans	227
5.13.2 Un premier problème	228
5.13.3 Un problème de logique formelle	232
5.13.4 Autres tâches données à GPS	238
5.13.5 Conclusion	243
6 Les programmes de jeux ; les études des psychologues	245
6.1 L'arborescence des coups légaux	246
6.2 L'évaluation des positions	253

6.3	Le <i>minmax</i> et le choix du coup à jouer	254
6.4	La procédure Alpha-Bêta	259
6.5	Les défauts fondamentaux liés au développement systématique de l'arbre des coups	262
6.5.1	Premier défaut : le manque total de stratégie.....	262
6.5.2	Deuxième défaut : l'effet d'horizon.....	263
6.6	Les études des psychologues sur la résolution de problèmes par les hommes	264
6.6.1	La mémoire	265
6.6.2	La perception	266
6.7	Les études des psychologues sur les joueurs d'échecs	267
6.7.1	La perception	267
6.7.2	Le raisonnement	269
6.8	ROBIN : Un programme d'échec intelligent	273
6.8.1	Un exemple d'analyse par ROBIN :	274
6.8.2	Différents types de plans aux échecs	276
6.8.3	Langage d'expression et d'exécution des plans :	277
6.8.4	L'optimisation de la défense : la notion de coup dangereux	279
6.8.5	Résultats	282
7	Les systèmes experts	287
7.1	Exemples de dialogue avec un système « expert ».....	289
7.1.1	Un diagnostic médical	289
7.1.2	Explication, par le système, de son propre raisonnement	289
7.2	Le fonctionnement du système MYCIN-TEIRESIAS	290
7.2.1	Le but	290
7.2.2	L'intérêt	290
7.2.3	Description.....	291
7.2.4	Explications fournies par MYCIN et dialogue en langue naturelle	295
7.2.5	L'acquisition de nouvelles connaissances : TEIRESIAS	296
7.2.6	Résultats et Analyse critique	301
7.3	Les systèmes de production	301
7.3.1	Description d'un système de production.....	303
7.3.2	Hypothèses et domaines d'applicabilité des systèmes de production	305
7.3.3	Variantes utilisées dans la conception des systèmes de production	306
7.3.4	Différents systèmes experts utilisant des règles de production	309
7.4	Les systèmes experts fondés sur la logique du premier ordre	319
7.4.1	Le Langage PROLOG	319
7.4.2	PEACE : un exemple de programme expert en PROLOG	321
7.4.3	Second exemple :	323
7.4.4	Intérêt de PROLOG et du Calcul des Prédicats	328
7.4.5	SNARK : un autre exemple de langage du premier ordre.....	330
7.5	La controverse déclaratif/procédural	336
7.5.1	Critique de l'informatique « procédurale » ou « les langages de programmation ne sont pas des langages »	337
7.5.2	Les structures de contrôle dans ces systèmes experts.....	338
7.5.3	Règles de production et procédures : exemples	339
7.5.4	Efficacité des systèmes experts	341
7.5.5	Conclusion sur la controverse procédural/déclaratif	345
7.6	Les différents types de connaissances et leurs représentations	345
7.6.1	Les différents types de connaissances	346
7.6.2	Les différentes représentations de la connaissance dans les systèmes actuels	347

7.6.3	Accès et modes d'utilisation des connaissances	348
7.6.4	Avantages et inconvénients des systèmes de production	349
7.7	La métaconnaissance	351
7.7.1	Méta-connaissance des objets de l'univers.....	351
7.7.2	Méta-connaissance des stratégies.....	352
7.8	Conclusion	353
7.9	Votre moteur de système expert en kit	356
8	ALICE	363
8.1	Description générale du système	364
8.2	Le langage ALICE	365
8.3	Le module de résolution de problème	370
8.3.1	La propagation des contraintes.....	371
8.3.2	Faire des choix.....	373
8.3.3	Résolution des exercices	375
8.4	Résultats	378
8.5	ALICE : exposé détaillé.....	378
8.5.1	La représentation des problèmes	378
8.5.2	Le traitement des contraintes algébriques	382
8.5.3	Les procédures de choix.....	386
8.6	Exemples détaillés de résolutions	392
8.6.1	Un casse-tête logico-arithmétique	392
8.6.2	Un problème de fabrication de papier	393
8.7	Table générale des problèmes traités par ALICE :	400
8.8	Généralité et efficacité.....	402
8.8.1	Exemple 1	403
8.8.2	Exemple 2	407
8.8.3	Exécution en ALICE	410
8.9	Saisie des énoncés en français	412
8.9.1	Principe de la traduction	412
8.9.2	Exemple de traduction	414
9	L'apprentissage	417
9.1	Différents types d'apprentissages	417
9.2	Apprentissage de paramètres au jeu de checkers	419
9.2.1	Apprentissage par cœur	419
9.2.2	Apprentissage livresque	420
9.2.3	Apprentissage sans professeur	421
9.2.4	Apprentissage généralisé	421
9.3	Apprentissage d'actions utiles par un robot : STRIPS	424
9.4	Apprentissage de règles	425
9.5	Apprentissage de plans	429
9.6	Apprentissage de caractéristiques	435
9.7	Apprentissage de concepts	437

Table des figures

2.1	Le cube de Necker	13
2.2	Quel(s) objet(s) représente ce dessin ?	13
2.3	Passer par chacun des neufs points en traçant au plus quatre segments de droite sans lever le crayon	13
2.4	Former quatre triangles équilatéraux avec six allumettes	13
2.5	Exemple 3	13
2.6	Solution dans le plan	13
2.7	Solution dans l'espace	13
2.8	Deux carrés de côté $a + b$ pour le théorème de Pythagore	15
2.9	Le damier, les noirs et les blancs.	16
2.10	Représentation d'une expression en arbre	29
2.11	Représentation d'une expression en liste.....	31
2.12	Le rôle de la représentation en mathématique	33
2.13	Représentation en machine des diagrammes de Venn.....	34
2.14	Construction de $\{A\}$ et $\overline{\{A\}}$	35
2.15	Un problème de construction géométrique	35
2.16	Un des graphes utilisés par DATE pour démontrer le théorème T2	36
2.17	Réseau sémantique« partitionné »	39
2.18	Les quatre cavaliers.....	40
2.19	Noms des cases et pas du cavalier à partir de A	41
2.20	Le parcours du cavalier A	41
2.21	Solution du problème des 4 cavaliers.....	41
2.22	Principe de l'évaluation en LISP : la fonction EVAL. Ce schéma suppose que 'L a toujours été auparavant développé en (QUOTE L). S1 désigne le premier fils de la S-expression S et S2 les éléments de S ensemble des frères de S1.....	46
2.23	Exemple de graphe	54
3.1	Décidabilité d'un système formel	62
3.2	Démonstration et valeurs de vérité.	64
3.3	Arbre de dérivation pour <i>MU</i>	67
3.4	Une abstraction du système formel DH.....	68
3.5	Abstraction du système formel sur <i>i</i>	69
3.6	La liste de Cantor	79
3.7	Second théorème de Gödel : la classe 1 n'est pas vide (Notation : $\vdash t$ signifie <i>t</i> est un théorème, $\nvdash t$ signifie <i>t</i> est un non-théorème).	84
3.8	Représentation et ordre de parcours de l'expression : $x^2 + (x + \sqrt{3})^2$ pour l'algorithme d'unification ; ordre antérieur ou préfixé : chaque symbole est traité dès qu'il est rencontré par descente de haut vers le bas et de la gauche vers la droite, (dans l'ordre des numéros de la figure de droite).	89

3.9	Représentation de $t \equiv y + \sqrt{3}$	92
3.10	Nouvelle expression E après substitution.....	92
3.11	Unification en logique des propositions.....	94
3.12	Unification (suite)	95
3.13	Unification (fin).....	95
3.14	Procédure générale de démonstration.....	98
3.15	Table de règles de réécriture en algèbre en vue de la simplification	99
3.16	Test géométriques	100
3.17	Test géométriques	102
3.18	Tests de Catell	106
3.19	Quelques problèmes de R. Bongard	107
3.20	Arbre de dérivation de : $(s \supset \neg t) \wedge t \mapsto \neg s$	109
3.21	Arbre de dérivation de $(a + b) + c \mapsto a + (b + c)$	110
3.22	Noeud OU en démonstration	111
3.23	Noeud ET en démonstration (on veut toutes les solutions)	112
3.24	Arbre de résolution de : $\sqrt{3} \cdot \tan x - 4 \sin^2 x = 0$	115
3.25	Arbre de résolution de $\sin(n+1)x + \sin(n-1)x - \sin 2x = 0$	116
3.26	Arbre de dérivation (a) et dérivation linéaire (b)	120
3.27	Notations dans les différentes langues.....	122
3.28	Mémorisation de l'arborescence de démonstration	128
3.29	Formé normale conjonctive et pile de résolution	130
3.30	Schéma de preuve par Résolution	134
4.1	Arbre de test pour un tri	144
4.2	Exemple de tri 1	145
4.3	Exemple de tri 2	145
4.4	Un problème de plus court chemin	146
4.5	Preuve de l'algorithme : le chemin ω	146
4.6	Graphe orienté pour un problème d'ordonnancement	150
4.7	Graphe réarrangé pour le problème d'ordonnancement	154
4.8	Les 7 ponts de la Pregel. Le problème du parcours eulérien	155
4.9	Quelques réductions établies entre NP-complets	173
4.10	Graphe résultant pour le problème de la K-clique	176
5.1	Parcours dans un labyrinthe	180
5.2	Méthode du gradient	181
5.3	Les flèches	181
5.4	Echecs des méthodes de gradients	182
5.5	Méthode du simplexe en programmation linéaire	184
5.6	Graphe réseau EDF	187
5.7	Recherche A^* pour un taquin	192
5.8	Le problème des dames, sur l'échiquier	195
5.9	Second choix	195
5.10	Deuxième second choix dans le problème des quatre dames	195
5.11	Les deux nouveaux premiers choix	195
5.12	La solution du problème des quatre dames	196
5.13	Arbre de recherche en énumération implicite	196
5.14	La solution du problème des quatre dames	199
5.15	Huit dames, version 2	200
5.16	Les huit dames	202
5.17	Les huit dames (suite)	203
5.18	Une solution complète : $[i]$: choix i , $[j]$: implication j	203

5.19 Impossibilité après trois choix	204
5.20 Fin de la recherche après deux choix	204
5.21 Graphe des décisions en programmation dynamique	206
5.22 Stratégie optimale : la sous-stratégie (a, c, b) ne saurait être meilleure que (a, d, b) , sinon (S_0, a, d, b, S_n) ne pourrait être optimale.	208
5.23 Carte de l'Europe.....	210
5.24 Graphe équivalent à la carte 5.23	210
5.25 Graphe pour le problème du colloque.....	211
5.26 Un contre-exemple.....	216
5.27 Deuxième contre-exemple	216
5.28 Le "circuit autour du monde" d'Hamilton	219
5.29 Matrice des distances pour un voyageur de commerce	219
5.30 Elimination d'axes trop coûteux.....	221
5.31 Solution partielle pour le voyageur	222
5.32 Résolution d'un problème du voyageur de commerce	226
5.33 Arbre de recherche de GPS.	228
5.34 Arbre de recherche de GPS pour l'exemple	231
5.35 Table de connexions pour GPS dans le problème de logique.....	234
5.36 Arbre de recherche de GPS en logique formelle	238
5.37 Les sept ponts de Königsberg.....	241
5.38 Séquence de lettres à compléter.	242
6.1 Exemple d'arborescence liée au raisonnement pour jouer un coup au jeu d'échecs.....	245
6.2 Le terrain au jeu de tic-tac-toe	247
6.3 Directions possibles au tic tac toe (ou aux échecs)	248
6.4 Trait aux noirs	250
6.5 Gain du premier joueur au tic-ta-toe	252
6.6 Une arborescence des coups légaux avec évaluation à profondeur variable des positions stables	256
6.7 Remontée des évaluations par <i>minmax</i> pour l'arborescence précédente ..	256
6.8 Coupe superficielle α	259
6.9 Coupe profonde α	260
6.10 Procédure $\alpha\text{-}\beta$ sur l'arborescence de la figure 6.6	261
6.11 Finale avec trait aux blancs	262
6.12 Effet d'horizon, H. Berliner (1974). Trait aux blancs	264
6.13 Evaluation de reconstitutions d'échiquiers (Le score est une fonction construite par De Groot qui tient compte du nombre de pièces sur la bonne case). (*) Joueurs de classe C : bons joueurs de clubs.	268
6.14 Position à reconstruire en 5 secondes (aux noirs de jouer)	269
6.15 Etude d'une position (De Groot, 1935)	270
6.16 Moyennes de quelques paramètres significatifs lors de l'étude d'une position d'échecs chez plusieurs joueurs.....	270
6.17 Arbre de preuve du coup $F \times Cd5$ dans la position <i>A</i> (de Groot 65)	273
6.18 E42 Trait aux blancs	274
6.19 M58 Trait aux noirs	277
6.20 M39 Trait aux noirs	280
6.21 M42 Trait aux blancs	282
6.22 B5 Trait aux blancs (Berliner)	284
6.23 T149 Trait aux noirs	285
7.1 Schéma de principe de MYCIN	291

7.2	Un dialogue avec TEIRESIAS	298
7.3	Exemple de dérivation en chaînage avant.....	302
7.4	Cycle de base d'un interprète de règles	303
7.5	Arbre de déduction en chaînage arrière avec les mêmes données qu'à la figure 7.3	304
7.6	Schéma général d'un système actuel de règles de production	305
7.7	Une représentation de haut niveau.....	317
7.8	Un circuit électrique pour PEACE	321
7.9	Transformations dans PEACE	322
7.10	Logique des sous-buts dans MECHO	326
7.11	Exemple de la troisième famille	328
7.12	Règle d'interprétation dans SNARK	333
7.13	Un objet dans la base de faits de SNARK.....	334
7.14	Tableau récapitulatif : SE et PP	339
7.15	Un système de règles de réécriture	339
7.16	Inversion de la chaîne de caractère MOT par le système de Markov précédent	340
7.17	Une règle complexe dans POLITICS (Carbonnel 78).....	342
7.18	Extrait du réseau d'inférences formé par l'interpréteur de PROSPECTOR à partir d'un jeu de règles de production classique : les flèches regroupent les hypothèses qui permettent d'arriver à une conclusion.	343
7.19	Exemple de réseau de Pétri : T1 peut se déclencher. P2 et P3 sont alors chargées. Ensuite le déclenchement de T2 et T3 charge P4 et P5 mais seule l'une d'entre elle pourra être active : puis le réseau se retrouvera dans l'état initial.....	344
7.20	Jeu de règles (Fig. 7.15) modifié avec indications de transitions.....	344
7.21	Représentation d'un système de règles de production sous forme de réseau de Pétri : les transitions ne sont pas indiquées par simplicité. Il y a deux sortes de jetons, vrais (V) ou faux (F), suivant la valeur de vérité de la partie gauche des règles.....	345
7.22	Représentation de connaissances	348
7.23	Exemple de méta-règle dans CRYSTALIS (Engelmore 79)	353
7.24	Cycle de base d'un système expert	358
7.25	Un moteur essentiel en logique des propositions	359
7.26	Jeu simplifié de règles pour le moteur zéro	360
8.1	Localisation des centres de secours	367
8.2	Effet d'un choix sur l'espace de recherche	373
8.3	Plan général de résolution pour ALICE	375
8.4	Optimisation de Processeurs.....	376
8.5	Localisation (représentation interne) : la région r_5 ne peut être déservie par e_2 , e_4 ou e_6	377
8.6	Schéma du graphe biparti de la représentation interne des problèmes	380
8.7	Fonction de Grundy	386
8.8	Schéma global de contrôle pour ALICE	386
8.9	Valeurs de trois critères pour 4 variables. Choix d'ALICE : variable 3	388
8.10	Restriction de l'espace de recherche	392
8.11	Solution optimale de coût 82	400
8.12	Arbre de recherche d'ALICE pour GERALD + DONALD = ROBERT. Une feuille marquée * correspond à une impossibilité.	407
8.13	Graphe des contraintes de monotonies	411

9.1	Différents types d'apprentissage	418
9.2	Représentation du damier.....	419
9.3	Deux évaluations de la même position	421
9.4	Hiérarchie de signatures à trois niveaux	423
9.5	Exemple de description d'opérateur élémentaire dans STRIPS	424
9.6	Séquences d'opérateurs dans STRIPS.....	425
9.7	Procédure d'apprentissage des règles.....	427
9.8	Un début de partie, trait aux blancs	429
9.9	Arbre de compréhension d'un coup aux échecs.....	431
9.10	Arbre d'étude d'une menace	431
9.11	Utilisation d'un plan (trait aux noirs)	434
9.12	Analyse et représentation d'une scène	435
9.13	Le concept d'arche.....	437
9.14	Exemples et contre-exemples d'un concept	441

Liste des tableaux

3.1	Preuve de $(p \supset q) \supset ((q \supset r) \supset (p \supset r))$ par la méthode des tables de vérité.....	74
3.2	Les théorèmes de limitation.....	86
3.3	Tableau (RNO) – Règles non obligatoires en trigonométrie.....	113
3.4	Tableau (RO) – Règles obligatoires (RO) en trigonométrie	114
3.5	Démonstration du théorème $(\neg p \supset p) \supset p$	125
3.6	Démonstration de la commutativité dans un groupe où : $\forall x \ x.x = e$, par la méthode d'Herbrand ou principe de Résolution	132
5.1	Méthodes de recherche heuristique	189
5.2	Coloration d'une carte de géographie	214
5.3	Coloration pour le problème du colloque	214
6.1	Algorithme <i>minmax</i>	258
8.1	Dictionnaire du langage ALICE	369
8.2	Syntaxe du langage ALICE	370

Avant-propos

La raison d'être de ce livre remonte sans doute à l'époque où j'étais lycéen ; comme tous les écoliers autour de moi j'ingurgitais le contenu des programmes scientifiques en maths et en physique en me demandant sans cesse : **à quoi cela peut-il bien servir ?** En physique, le rapport avec la réalité se voulait évident, mais nous n'arrivions pas à y croire ; trop de choses restaient mystérieuses. De plus, chaque année, le nouveau professeur nous expliquait que le modèle de l'année précédente était faux... La physique était perçue comme un jeu bizarre du monde des adultes. En mathématiques, la situation était différente : la beauté de l'abstraction, tout autant que le plaisir de trouver une démonstration élégante étaient motivants. Un sentiment profond de duperie venait cependant gâcher tout cela : on nous assénait sans cesse des définitions et des preuves comme des réalités révélées ; **le pourquoi des choses n'était jamais donné.** La plupart des preuves paraissait sortir par magie du bâton de craie du professeur. Comment pouvait-on enchaîner toutes ces lignes et penser dès le début à l'aboutissement merveilleux de la dernière d'entre elles ? Et par dessus tout, « **à quoi cela pouvait-il servir** » ?

En réalité cela devint clair, plus tard, après quelques années de vie « active »— **tout cela ne sert à rien**, ou, du moins, ne sert pas directement : tous ces sujets d'études sont arbitrairement choisis et placés dans les programmes scolaires. En vérité, ce ne sont que des prétextes pour faire passer quelque chose de plus noble : apprendre à comprendre, apprendre à résoudre des problèmes, **apprendre à apprendre**. Mais, curieusement, ces sujets-là, justement, ne sont nullement avoués et guère enseignés. Une certaine forme de terrorisme intellectuel classe définitivement une bonne part des écoliers « nuls en maths », alors que leur seul tort est de ne pas comprendre... ce qui n'a jamais été dit. D'autres s'en tirent parce qu'ils ont entrevu plus tôt la règle du jeu implicite. D'autres encore apprennent tout par coeur...

Or, il existe actuellement un domaine de recherche, où l'intention première des chercheurs est précisément de comprendre comment un système de traitement de l'information – homme ou machine – peut assimiler, analyser, transposer, généraliser ce qui lui est enseigné et, par là, faire face à des situations concrètes et résoudre des problèmes.

Le nom de cette discipline est **l'intelligence artificielle** ; c'est la fille ainée de l'informatique : elle a pour sujet d'étude toutes les activités intellectuelles de l'homme pour lesquelles aucune méthode n'est a priori connue. C'est ainsi que son objet a pu être défini par « tout ce qui n'a pas encore été fait en informatique ». Si l'informatique est la science du traitement de l'information, l'intelligence artificielle – I.A. dans la suite – s'intéresse à tous les cas où ce traitement ne peut être ramené à une méthode simple, précise, algorithmique.

Ces cas sont innombrables, même dans des situations très banales comme la lecture d'un texte, tel celui-ci. Le même caractère, le | par exemple sera perçu, par le système visuel, suivant le contexte, tantôt comme un tantôt comme un i, tantôt comme un signe de valeur absolue... Le cas de l'écriture manuscrite est pire encore : les trois mots suivants utilisent ainsi le même graphème **w** :

motif note utile

Comment notre système cognitif lève-t-il t'il l'ambiguïté ? Tel est un des problèmes clés soulevés en I.A. L'identification de visages humains (qui sont moins aisés à caractériser formellement que de simples lettres), la compréhension de textes (et non plus de lettres

isolées), celle de paroles (sans trace écrite), la démonstration de théorèmes, la résolution de problèmes, le choix d'un coup au jeu d'échecs, la construction d'un emploi du temps, la réponse à un test de quotient intellectuel, la conception d'un plan en architecture, l'élaboration d'un diagnostic en médecine, l'analyse, d'un article de journal, sont autant de sujets d'étude dans cette discipline. Ils ont tous fait l'objet de réalisations récentes souvent impressionnantes.

Chapitre 1

Intelligence artificielle

Les principaux sujets d'étude de l'Intelligence artificielle sont l'acquisition et la représentation de la **connaissance** sous toutes ses formes.

L'objectif des chercheurs en I.A. n'est pas seulement d'élaborer de nouvelles théories, mais également de concevoir et de mettre en oeuvre des programmes aussi généraux que possible.

Nous examinerons tout d'abord le rôle joué par l'informatique traditionnelle. Nous verrons ensuite pourquoi et comment les objectifs de l'I.A. s'écartent de cette démarche classique pour apporter un renouveau fondamental.

1.1 Statut de l'informatique

L'informatique est utilisée de deux façons différentes.

1. En tant que support pour une modélisation : c'est alors un prolongement des mathématiques qui met l'accent sur le phénomène temps et les processus évoluant dans le temps. La solution d'un problème ayant été analysée mathématiquement, elle est traduite par un programme exécutable par un ordinateur.
2. En tant qu'outil concret de vérification d'hypothèses : un ordinateur est, par construction, impartial et fidèle ; c'est une merveilleuse machine pour **tester des idées**.

Ainsi, l'informatique, et plus encore l'I.A., entretiennent-elles des liens privilégiés avec la linguistique, la psychologie et la logique. Ces trois disciplines, en effet, s'intéressent aux phénomènes cognitifs, à la compréhension, au raisonnement.

Il est remarquable que ces liens jouent dans les deux sens : premièrement, aujourd'hui, les linguistes, les psychologues, les logiciens **programment** les modèles nouveaux qu'ils construisent (les biologistes, les médecins, les mathématiciens tendent à faire de même). Deuxièmement, les chercheurs en I.A. reprennent ces **modèles** et tentent d'en déduire des **logiciels** effectifs de résolution de problèmes.

Il existe des rapports particulièrement étroits entre l'I.A. et les sciences cognitives : la raison en est simple : à part l'ordinateur, le seul outil à raisonner auquel nous ayons accès est notre cerveau, et encore ce dernier est-il d'un abord difficile, pour ne pas

dire impossible. Toute introspection gène l'expérience en cours. L'ordinateur fait donc figure d'auxiliaire extraordinaire.

Du même coup, l'I.A., après les révolutions fondamentales de Copernic et Darwin, entraîne une remise en cause de la place de l'homme dans la nature. C'est bien, en effet, ici le monopole de l'intelligence qui lui est contesté.

1.2 L'intelligence artificielle

L'I.A. est une science qui date d'une trentaine d'années. Son objet est de reconstituer à l'aide de moyens artificiels – presque toujours des ordinateurs – des raisonnements et des actions intelligentes. Les difficultés sont a priori de deux types :

1. Pour la plupart de nos activités nous ne savons pas nous-mêmes comment nous nous y prenons. Nous ne connaissons pas de méthode précise – pas d'algorithme disent aujourd'hui les informaticiens – pour comprendre un texte écrit ou reconnaître un visage, démontrer un théorème, établir un plan d'action, résoudre un problème, apprendre...
2. Les ordinateurs sont a priori très loin d'un tel niveau de compétence. Il faut les programmer depuis le tout début. Les langages de programmation ne permettent, en effet, que d'exprimer des notions très élémentaires.

L'I.A. est, de ce double point de vue, une science expérimentale : expériences sur ordinateurs qui permettent de tester et d'affiner les modèles exprimés dans les programmes sur de nombreux exemples ; observations sur l'homme (en général le chercheur lui-même) pour découvrir ces modèles et mieux comprendre le fonctionnement de l'intelligence humaine.

Dans le premier paragraphe nous préciserons la définition et les domaines de l'I.A. Nous relaterons ensuite l'histoire de cette science nouvelle en mettant en évidence les lignes de force qui ont guidé les chercheurs. En dernier lieu, nous dresserons le bilan des méthodes actuelles, des résultats acquis aujourd'hui et des travaux pour demain.

1.3 Définition et domaines de l'intelligence artificielle

1.3.1 Définition

Tout problème pour lequel aucune solution algorithmique n'est connue, relève a priori de l'Intelligence artificielle.

Par *algorithme*, il faut entendre ici toute suite d'opérations ordonnées, bien définies, exécutables sur un ordinateur actuel et qui permet d'arriver à la solution en un temps raisonnable (de l'ordre de la minute ou de l'heure). Ainsi, on ne connaît pas d'algorithme pour jouer aux échecs, bien que ce jeu ne comporte qu'un nombre fini de situations, mais les étudier toutes demanderait des millénaires. De même, il n'existe pas d'algorithme pour effectuer un diagnostic médical, résumer un texte ou le traduire dans une langue étrangère.

1.3.2 Domaines

Ces domaines – très variés – où nous agissons sans méthode absolument définie, possèdent en commun deux caractéristiques remarquables :

- ils concernent des informations **symboliques** : lettres, mots, signes, dessins. Ceci s'oppose aux traitements numériques habituellement confiés aux ordinateurs,
- ils impliquent des **choix**. En effet, dire qu'il n'existe pas d'algorithme, c'est dire qu'à certains pas, il faut choisir sans certitude entre plusieurs possibilités. Ce non-déterminisme fondamental, cette liberté d'action est une **composante essentielle de l'intelligence**.

Le premier problème auquel se heurte la recherche en I.A. est celui de la saisie de l'information. Les technologies et les logiciels actuels sont loin de pouvoir rivaliser avec les capteurs et effecteurs humains en vision, manipulation, goût, odorat, compréhension ou émission de parole.

1.3.2.1 La perception et la reconnaissance des formes

Un système de traitement de l'information reçoit des données de ses capteurs. De nos cinq sens, l'oeil – et le système visuel – est sans doute le plus important. Les caméras et les lasers sont couramment connectés aujourd'hui à des programmes de « reconnaissance d'images » et « d'analyse de scènes ». Des microphones servent d'autre part à capter des sons.

Ce domaine du codage et du traitement des signaux est connu sous le nom de **reconnaissance des formes**. C'est un point de passage obligatoire pour un système autonome de traitement de l'information. Mais, en réalité, les problèmes qui se posent en I.A. sont loin d'être résolus, même si l'information est déjà codée et mémorisée. Aussi, ce sont aux problèmes de compréhension et de raisonnement qui viennent logiquement ensuite, que s'attaquent spécifiquement les travaux d'I.A.

1.3.2.2 Les mathématiques et la démonstration automatique de théorèmes

L'I.A. attache, par définition, une importance particulière à **l'information symbolique**, c'est à dire non numérique. Ainsi, les premiers domaines où les chercheurs ont travaillé furent les mathématiques et les jeux. Dans les deux cas, en effet, ces domaines, déjà bien formalisés, et qui, en outre, font figure de bastion de l'intelligence humaine, sont d'excellents sujets tests.

Les premiers programmes en démonstration automatique virent le jour en 1957, soit 10 ans après l'apparition des premiers ordinateurs. Ils travaillaient d'abord sur des théories limitées, puis, de plus en plus riches. Le niveau de l'individu moyen fut rapidement dépassé. Mais celui du bon mathématicien n'est toujours pas atteint aujourd'hui ! La théorie de la preuve et celle des méthodes effectives de démonstration furent étudiées de plus près et plusieurs développements furent apportés. De plus, les formalisations

mathématiques de type logique se révélaient indispensables pour les applications fondamentales : robotique, résolution de problèmes, interrogation de bases de données. Ce sujet reste ainsi l'un des domaines phare en I.A.

1.3.2.3 Les jeux

Plus encore que les systèmes formels des mathématiques, les jeux concernent des univers limités, aux règles bien spécifiées et pourtant riches en possibilités déductives. C'est pourquoi ils furent et sont encore un sujet de prédilection en I.A. Ici encore, le niveau du joueur de club moyen fut aisément dépassé, tandis que celui du champion du monde ne l'est pas encore. De façon inattendue, les difficultés rencontrées à propos des jeux furent, en effet, les mêmes qu'en mathématiques et que dans beaucoup d'autres domaines. Elles ont toujours trait à la **masse considérable de connaissances** que l'homme a su accumuler à travers les âges. Pour les jeux de hasard, tel le poker ou le backgammon, où le raisonnement s'efface devant le calcul des probabilités, les performances des programmes sont, en revanche, remarquables.

1.3.2.4 La résolution de problèmes

Le mot résolution est à prendre dans son sens large. Il s'agit de poser, d'analyser et de représenter des situations concrètes, tout autant que de les résoudre. Le bilan, très restreint, des problèmes aujourd'hui bien résolus une fois fait. Il reste le vaste champ de tous les autres. C'est la capacité d'invention et de généralisation qui est ici en cause, qu'il s'agisse de problèmes de la vie de tous les jours, de recherche opérationnelle ou de mathématiques.

Les robots doivent en particulier être capables de résoudre des problèmes dont la solution nous vient inconsciemment : monter sur quelque chose pour saisir un objet, allumer la lumière pour y voir clair.

1.3.2.5 Compréhension du langage naturel (LN)

Dans la partie « compréhension du langage naturel », les chercheurs s'intéressent à l'analyse et à la génération de textes, à leur représentation interne, à la mise à jour des connaissances nécessaires à leur compréhension : connaissances syntaxiques, sémantiques, pragmatiques. Cette partie n'est pas traitée dans ce livre : Cf : Pitrat J. (1985) : *Textes, ordinateurs et compréhension*, Eyrolles.

Les retombées et les implications sociales des recherches en I.A. seront bientôt importantes.

Ainsi, plusieurs disciplines sont désormais directement concernées : la psychologie (le cerveau humain reste le point de passage obligatoire pour faire de l'I.A.), la logique, la linguistique, la biologie (modèles de transmission d'information par les gènes), l'informatique (systèmes d'exploitations évolués, interrogation de bases de données, programmation automatique), la médecine (aide au diagnostic) et surtout peut-être la didactique et la pédagogie dans toutes les disciplines (le niveau de détail des pro-

grammes d'I.A. met en relief certaines carences des enseignements traditionnels et met en évidence tout ce qui n'est pas assez enseigné).

1.4 Historique

Les ordinateurs, s'ils ne sont pas indispensables pour construire et tester les modèles d'I.A., sont un moyen de recherche formidable et, concrètement, c'est avec eux que l'I.A. a pris son essor. Dès 1954, A. Newell fait le projet d'écrire un programme pour bien jouer aux échecs. C. Shannon, père de la théorie de l'information, avait déjà proposé une méthode pour cela. A. Turing, un des premiers informaticiens, l'avait affinée et simulée à la main. A la Rand Corporation, J. Shaw et H. Simon s'associent au projet de Newell. Ils s'assurent la collaboration d'une équipe de psychologues d'Amsterdam, dirigée par A. de Groot, qui avait étudié de grands joueurs humains. Un langage est tout spécialement créé par cette équipe pour manipuler facilement en machine les informations symboliques, les pointeurs, les listes chaînées : c'est IPL1 (Information Processing Language 1, 1956) qui sera le père de LISP (J. Mac Carthy 1960). Le premier programme d'I.A. est finalement le « LOGIC THEORIST » qui travaille en logique des propositions et donne **sa première démonstration le 9 août 1956**.

Le programme d'échecs NSS (Newell, Shaw, Simon) voit le jour en 1957. Sa structure et celle du LOGIC THEORIST. Les notions de « situation souhaitable » et d'*« heuristiques »* (du grec euriskein : trouver = règle qui permet de faire un choix en l'absence de résultat théorique sûr) conduisent, un peu plus tard, à la conception de GPS, « GENERAL PROBLEM SOLVER ». Ce programme, par analyse de différences entre situations et par construction de buts, sait résoudre aussi bien le casse tête des « Tours de Hanoï », qu'intégrer formellement une expression.

Les informaticiens s'intéressent tous alors à l'I.A., certains écrivent déjà des essais célèbres, J. Mac Carthy, M. Minsky, H. Simon. D'autres programmes voient le jour. Gelernter (1960), à propos de géométrie, constate qu'un programme peut mieux faire que son programmeur ! Pour démontrer que le triangle ABC, qui a les deux angles à la base B et C égaux, a deux côtés égaux, le programme, au lieu de la démonstration classique des livres, qui consiste à construire la hauteur en A, applique simplement un théorème d'égalité des triangles ABC et ACB ! Le résultat est évident... Le programme EPAM (Elementary Perceiving and Memorizing Program) est conçu par E. Feigenbaum comme une simulation psychologique.

Le traitement du langage est entrepris très tôt avec une application sur la recherche d'information dans les bases de données : BASEBALL (Green et al 1961) répond aux questions sur les matches. La résolution de problèmes d'algèbre, posés en anglais, est à la portée du système STUDENT (Bobrow 1964).

Les ambitions en traduction automatique sont grandes. Les projets dans ce domaine occupent de nombreuses équipes. Les chercheurs croient alors avant tout à l'analyse syntaxique et à l'information donnée par les dictionnaires (méthodes par **mots-clés**).

Cela ne suffit pas, comme de sévères rapports le mettront en lumière (Dreyfus 1972, Lighthill 1973), mais les chercheurs mettront des années à se rendre compte que la traduction automatique n'est pas un problème en soi, et qu'elle passe nécessairement par la **compréhension**.

Un cadre logique nouveau, construit à partir d'une systématisation du raisonnement

par l'absurde, naît en 1965 (J. Robinson). Il permet la formalisation de nombreux problèmes et leur interprétation en machine, il est utilisé aussi bien pour démontrer des théorèmes (Slagle, Green, Kowalski), que pour vérifier des programmes (King, Waldinger), manipuler des objets (Nilsson, Fikes). Il est le point de départ d'un langage de programmation original : PROLOG, qui a la puissance de la logique du premier ordre, et a été conçu par A. Colmerauer en 1971. (Cf chapitre 3).

De façon constante, les recherches en I.A. sont ainsi jalonnées par des générations de langages et de systèmes, de plus en plus généraux, qui rapprochent la programmation des ordinateurs de notre raisonnement et de notre vocabulaire habituel. Outre LISP et PROLOG, citons les plus importants : PLANNER et QA4 permettent à l'aide des concepts de but et d'assertion de modéliser et de formaliser la déduction en résolution de problèmes ; MACSYMA et REDUCE permettent la manipulation formelle des expressions mathématiques ; TMS permet de gérer des informations non sûres et de tester leur cohérence.

Les chercheurs en robotique utilisent, par ailleurs, ces résultats au fur et à mesure, pour piloter des systèmes fixes ou mobiles dans l'univers réel. Ils font face en outre à tous les problèmes de capteurs. La vision par caméra, la détection des contours (Gussman, Waltz, Winston) et d'objets cachés, dans des univers complexes, sont des problèmes de mieux en mieux résolus.

Mais, jusqu'à cette époque (1968), les chercheurs ont essentiellement utilisé des micro-univers de travail : jeux, géométrie, calcul intégral, monde de blocs, phrases courtes sur petits vocabulaires. La méthode retenue est presque toujours la même ; c'est **l'amélioration du combinatoire** : on restreint l'énumération exhaustive à l'aide du « bon sens », de fonctions d'évaluation numériques et d'heuristiques.

L'honnête homme, certes, ne se sert pas que de cela, mais des dizaines de réalisations ont été programmées dans de très nombreux univers dont les performances rivalisent effectivement avec les nôtres (les micro-ordinateurs de jeux – Echecs, Dames, Othello, Bridge, Go – tombent dans cette catégorie). Il reste mieux à faire : atteindre le comportement des **experts**.

1.5 Résultats

Le début des années 70 marque une réelle transition dans les recherches en I.A. et ceci pour deux raisons.

En premier lieu, tous les chercheurs se sont, petit à petit, convaincus que tous les programmes précédents manquaient beaucoup d'une composante essentielle : la connaissance approfondie du domaine concerné. Ce qui fait la différence entre l'expert et l'homme de la rue, c'est **l'expérience**, autrement dit la connaissance accumulée au fil des années. Si l'on désire améliorer les performances d'un programme d'I.A., ce n'est plus sur l'ajustement des heuristiques ou des coefficients numériques qu'il faut travailler, mais bien sur la **mise en oeuvre de raisonnements, et d'expériences symboliques**.

En second lieu, se pose un problème immédiat et concret : **comment donner cette connaissance à un programme** alors que le programmeur lui-même ne la possède pas ?

La réponse est claire : le programme doit l'acquérir en donnée, de l'expert lui-même. Les

chercheurs ont été ainsi conduits à doter les systèmes d'I.A. d'une possibilité que n'ont pas les langages de programmation standards : les programmes d'I.A. doivent pouvoir saisir une information comme « Il fait beau, à Paris, ce 10 février », la stocker et ne l'utiliser qu'à bon escient. Il y a ici séparation entre l'énoncé d'un fait et la manière de se servir de ce fait. Par opposition, un langage de programmation ne permet d'exprimer que des ordres **exécutables**.

Cette caractéristique est vitale puisque l'expert fournit des faits isolés, sans pouvoir dire par avance à quel moment il conviendra de les prendre en compte.

Les recherches en résolution de problèmes et en compréhension du langage naturel convergent alors vers cette question centrale de la **représentation des connaissances**.

Plusieurs programmes voient le jour vers 1970, qui sont fondés sur ces idées. Le premier est DENDRAL. Il déduit des formules chimiques développées de corps chimiques à partir de leur spectrographe de masse. Mis au point à Stanford avec l'aide de J. Lederberg, prix Nobel, il a été progressivement nourri de l'expérience de celui-ci. Il contient plusieurs milliers de faits élémentaires. C'est l'un des premiers *systèmes experts*. Ces faits sont donnés sous forme de règles granulaires dans le langage même de l'expert. Ce système a des performances remarquables et est actuellement vendu avec le spectromètre par le fabricant.

Bien sûr, l'idéal est que le programme déduise lui-même les règles à partir de son expérience, en un mot, qu'il apprenne. C'est ce qui a été réalisé par l'équipe DENDRAL au SRI (Stanford Research Institute) : METADENDRAL, par simulations répétées et à l'aide de quelques règles générales indiquant les familles de coupures possibles, infère progressivement les règles particulières pour chaque liaison chimique, puis pour chaque corps. Il a ainsi effectué, de toutes pièces, l'analyse de familles chimiques mal connues et permis ainsi la rédaction d'articles dans des revues internationales de chimie.

Terry Winograd est, quant à lui, l'auteur de SHRDLU (1971), un robot manipulateur de cubes, qui dialogue en anglais. Le système ne s'intéresse plus seulement à la syntaxe des phrases mais en comprend véritablement le sens grâce à des connaissances **sémantiques** et **pragmatiques** sur son univers de blocs. Il sait lever les ambiguïtés (par exemple les références pronominales), résoudre les métaphores, justifier son comportement et rendre compte de ses actions. En outre, il montre, sur l'univers réel, que tout ceci est bien géré par le programme.

L'I.A. arrive aujourd'hui à un stade opérationnel.

Les chercheurs à plein temps ne sont que quelques centaines au monde (dont une centaine en France), mais leurs résultats concernent chacun d'entre nous. Les médias en parlent beaucoup et annoncent souvent les « robots » pour demain. Il faut, en fait, bien comprendre que cette recherche est lente et difficile : car, loin de chercher à inventer des recettes miracles (des algorithmes), les chercheurs en I.A. s'efforcent de reconstituer peu à peu et de mettre en machine l'expérience et la connaissance des spécialistes dans tous les domaines.

Or, en général, cette **information n'est pas disponible** : il faut longuement interroger l'expert et retrouver ce qu'il a mis inconsciemment des années à apprendre. Pour cette tâche, ont été développés des langages et des systèmes de représentation. Mais il faut *rentrer* plus d'informations que ce que contient un dictionnaire ou une encyclopédie. La tâche n'est pas impossible avec les outils et les méthodes déjà acquis, mais elle est de longue haleine. Heureusement elle est passionnante, car elle permet d'apprendre beaucoup sur **l'homme** lui-même et son intelligence ; c'est bien l'homme,

en effet ici, le sujet d'étude fondamental et il est à parier que lorsque la tâche sera accomplie, les programmes d'I.A. du prochain siècle pourront tourner sur les ordinateurs d'aujourd'hui.

Chapitre 2

Représenter un problème

Ce chapitre constitue une introduction et un plan pour toute la suite du livre.

2.1 Introduction

Une personne ressent un **problème** dès qu'elle se trouve dans une situation où elle désire quelque chose alors qu'elle ne voit pas immédiatement la suite d'actions à accomplir pour obtenir ce quelque chose.

Le mot problème vient du grec « *βαλλειν* » : jeter : un problème est un « objet jeté en avant (ce mot est de la même famille que bal, parabole, hyperbole, symbole, chablis (bois jeté à terre où l'on a fait ensuite pousser la vigne !) ». Outre les problèmes que chacun peut rencontrer dans la vie de tous les jours, les premiers problèmes « imposés » sont ceux rencontrés dès l'école primaire. Leur caractéristique essentielle, qui se perpétue pendant toute la scolarité, est d'être des problèmes « tout posés », donnés dans un langage conventionnel. Ce langage est le plus souvent celui des **mathématiques** ; il nous marque tous profondément, la compréhension plus ou moins grande de ce langage, de ses règles du jeu, de ses informations implicites, peut créer très tôt un clivage entre les élèves, clivage qui distingue les élèves à « l'esprit de finesse » et ceux à « l'esprit de géométrie ». L'objet du second paragraphe de ce chapitre, qui retrace brièvement l'histoire de ce langage mathématique, est de montrer que le sujet n'est pas simple et que l'homme a mis des millénaires à forger, après de multiples tâtonnements, les concepts utilisés aujourd'hui et que l'aventure n'est certes pas terminée.

Mais, la plupart des **problèmes** rencontrés en dehors de l'enseignement primaire ou secondaire, **ne sont pas tout posés**. Ils nous arrivent de façon inopinée : ils ne sont, le plus souvent, que partiellement décrits ; ils sont, bien des fois, énoncés par une tierce personne qui utilise divers moyens pour transmettre l'information, diverses *représentations* de l'univers réel : voix et intonation, gestes, dessins. Le langage naturel est ici un véhicule privilégié. Il n'en comporte pas moins, du point de vue de la résolution de problèmes, quatre défauts essentiels : le langage naturel est **incomplet, redondant, ambigu, incorrect**.

2.2 Le langage naturel

En effet, il est tout d'abord clair qu'une grande partie de l'information n'est **pas explicitée** dans un dialogue habituel : les deux interlocuteurs sont supposés posséder une même connaissance globale du sujet ; mais, lorsque cette hypothèse n'est pas vérifiée, une interprétation entièrement erronée n'est pas exclue. Cet état de fait est à l'origine de bien des déboires en informatique. Les « cahiers des charges », utilisant des méthodes rigoureuses de « spécifications », luttent aujourd'hui contre ce défaut du langage naturel.

En second lieu, la **redondance** est couramment utilisée pour insister sur les points qui paraissent importants à celui qui pose le problème : mais rien ne dit que les vraies difficultés lors de la résolution correspondront à ces points là.

En troisième lieu, le langage est continuellement ambigu, sinon il nous serait simplement impossible de communiquer ; mais, comme pour l'incomplétude, une erreur lors de la levée d'une ambiguïté, peut conduire à une interprétation complètement fausse. Enfin, le langage, parlé notamment, est **grammaticalement incorrect** ; c'est une évidence, mais paradoxalement c'est là sans doute le moindre défaut du langage naturel.

2.3 Poser un problème

Poser un problème, que celui-ci soit issu d'un manuel scolaire, d'une expérience personnelle ou de celle d'une tierce personne, consiste tout d'abord à **comprendre l'énoncé**, c'est-à-dire à en éliminer l'incomplétude, la redondance et l'ambiguïté. En d'autres termes, cela revient à trouver une représentation où tout est dit. Un deuxième type de représentation, distincte du langage, intervient souvent à ce stade : il s'agit des représentations graphiques. **Le système visuel** est en effet, chez l'homme, un formidable outil de saisie et de traitement de l'information. Un problème n'est véritablement compris que lorsqu'une telle représentation a été trouvée dans laquelle tous les éléments intervenant dans le problème sont représentés sans redondance ni ambiguïté. L'univers de recherche de la solution est alors bien délimité et le plus souvent la difficulté principale du problème apparaît déjà. Une bonne partie de la **pragmatique** et de la **sémantique** du problème a alors été traduite formellement ; le problème est devenu à la fois plus abstrait et plus pur ; on parle alors *d'énoncé fermé*.

2.4 Les énoncés fermés

La forme mathématique la plus générale d'un énoncé de problème peut s'écrire :

Trouver dans un ensemble X donné, les points x satisfaisant un ensemble de contraintes données $K(x)$.

Exemple (problème posé par A. Binet à Inaudi) : « Trouver dans l'ensemble des entiers naturels les x tels que : $x^3 + 84 = 37x$ »

Remarques :

- a) La donnée de l'espace X correspond généralement à la donnée simultanée – mais implicite – de la structure de X et des **opérations licites** sur X . La connaissance de X est un condensé crucial d'information.
- b) Le premier énoncé fermé obtenu peut être, en général, traduit à nouveau pour, tenant compte des contraintes $K(x)$, réduire l'espace X et améliorer ainsi la représentation du problème. A la limite, c'est par une suite de changements de représentations que le problème est finalement résolu, l'énoncé fermé final livrant directement la solution.

Il existe deux variantes principales à cette forme de base d'un énoncé fermé :

Variante 1 :

L'univers comporte une situation initiale donnée, soit S_0 , une situation finale donnée, soit S_1 , et une liste finie donnée d'opérateurs, O_{ab} , qui permettent de passer d'une situation, S_a , à une autre, S_b . Il s'agit de trouver un chemin de S_0 à S_1 . Exemple : le taquin de Sam Loyd.

$S_0 =$	<table border="1"> <tr><td>2</td><td>□</td><td>6</td><td>7</td></tr> <tr><td>12</td><td>9</td><td>10</td><td>3</td></tr> <tr><td>15</td><td>5</td><td>1</td><td>8</td></tr> <tr><td>4</td><td>13</td><td>14</td><td>11</td></tr> </table>	2	□	6	7	12	9	10	3	15	5	1	8	4	13	14	11	$S_1 =$	<table border="1"> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>5</td><td>6</td><td>7</td><td>8</td></tr> <tr><td>9</td><td>10</td><td>11</td><td>12</td></tr> <tr><td>13</td><td>14</td><td>15</td><td>□</td></tr> </table>	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	□
2	□	6	7																																
12	9	10	3																																
15	5	1	8																																
4	13	14	11																																
1	2	3	4																																
5	6	7	8																																
9	10	11	12																																
13	14	15	□																																

Opérateurs O_{ab} : on peut passer d'une situation à une autre en faisant glisser une case adjacente à la case vide, dans la case vide. Un problème énoncé dans cette variante se ramène au premier cas en posant :

$X = (\text{séquences d'opérateurs})$.

Une solution x est en effet une séquence particulière

$x = (O_{ab}, O_{bc} \dots O_{tu}, O_{uv})$ avec $S_a = S_0$ et $S_b = S_1$

L'ensemble des contraintes $K(x)$ est exprimé dans la loi de succession des opérateurs (la situation terminale de l'un est situation initiale pour le suivant).

Variante 2 :

Il s'agit de la formulation classique d'une démonstration en mathématiques :

Démontrer $C(x)$ à partir des hypothèses $H(x)$.

Exemple : Démontrer pour tout n , $n \in \mathbb{N}$

$$\sum_{i=1}^n i^3 = (\sum_{i=1}^n i)^2$$

Cette variante se ramène à la précédente, et par la même au premier cas, en posant : $S_0 = H(x)$ et $S_1 = C(x)$. Une différence notable intervient cependant dans la mesure où les **opérateurs** de passage d'une situation à une autre **ne sont pas donnés**. L'art du mathématicien est de retrouver ceux qui lui sont utiles sur l'énoncé particulier. En outre, il advient couramment que la situation finale ou « conclusion » $C(x)$ ne soit pas non plus donnée. Le problème ci-dessus prend alors la forme : « calculer $\sum_{i=1}^n i^3$ ». Le problème reste alors parfaitement ambigu car le critère d'arrêt est subjectif. On attend une expression « plus simple » que celle de départ, mais la simplicité d'une expression n'est pas un concept mathématique rigoureux.

2.5 La démarche générale en résolution de problème

La démarche de l'esprit humain en face d'un problème semble devoir passer dans une vue quelque peu idéale des choses – par sept étapes clés :

- E1** : Saisie de l'énoncé
- E2** : Inférences immédiates
- E3** : Jeu avec la situation
- E4** : Incubation
- E5** : Choix d'une meilleure représentation : énoncé fermé
- E6** : Résolution partielle (retour à E2) ou totale
- E7** : Vérifier, généraliser

- La première étape E1 est liée à nos capteurs : l'audition et la vision y jouent, comme il a déjà été souligné, des rôles fondamentaux. De plus, l'homme possède des capacités réduites de mémorisation immédiate (cf. section 6.6) qui entravent la saisie des énoncés longs.
- L'étape E2 fait appel aux connaissances générales du sujet pour :
 - compléter les informations manquantes.
 - résumer de longues phrases par des notations appropriées et plus manipulables. Dessins, graphes, contraintes algébriques sont ici formés.
- Remarque : Dès 1962, des programmes simples fondés sur les « mots clés » étaient capables de résoudre des exercices posés en anglais dans le domaine de la cinétique (Bobrow 62) ou des probabilités (Gelb 64).
- L'étape E3 est, chez l'homme en tout cas, capitale. Il s'agit tout d'abord, par retour en E1 et E2, de vérifier que rien ne manque, qu'il n'existe pas d'erreur grossière d'interprétation. Il s'agit, en second lieu, de voir « où git le lièvre », c'est-à-dire où est la difficulté du problème.
- L'étape E4 est tout aussi primordiale. Il nous est, à ce stade, nécessaire de nettoyer notre mémoire des faits parasites liés à la formulation initiale, pour laisser venir les bons opérateurs sur le nouvel « énoncé fermé ». Notons qu'une méthode particulière en E4 consiste à **laisser le problème de côté** et à faire autre chose pendant un temps. Nous sommes, en effet, ainsi programmés que l'oubli est un phénomène automatique excellent pour chasser les anciennes idées et faire apparaître les nouvelles ; Se « **distraire** » du problème en cours est ainsi une excellente méthode pour sa résolution.
- L'étape E5 doit marquer la formulation d'un énoncé fermé, c'est-à-dire complet, non ambigu, et non redondant. La difficulté du problème peut ici être évaluée plus précisément : la taille de l'espace de recherche X , la complexité des contraintes $K(x)$ et le nombre des opérations légales sont de bons indicateurs.
- Au cours de l'étape E6, c'est presque toujours une meilleure représentation qui est recherchée ; celle-ci est définie par un espace de recherche plus petit. Un nouvel énoncé fermé est obtenu et le cycle peut recommencer à partir de l'étape E2.
- L'étape E7 marque la fin de la résolution du problème proprement dit. Il n'est pas inutile de revenir en E1 – voire de discuter avec celui qui a posé le problème – pour vérifier l'adéquation de la solution à l'énoncé de départ, vérifier le comportement de la solution dans les cas limites, jauger les paramètres d'entrée importants pour la valeur de la solution finale. Par ailleurs, il est intéressant de tester la généralité de la démarche suivie sur le problème en cours :
 - Peut-on généraliser le problème donné en conservant la méthode employée ?

- Existe-t-il d'autres problèmes où la même méthode s'appliquerait avec succès ?
- Existe-t-il d'autres méthodes pour le même problème ?

Etape 1 : Saisie de l'énoncé et Etape 2 : Inférences immédiates.

Donnons tout d'abord quatre situations simples qui peuvent être perçues différemment selon les individus.

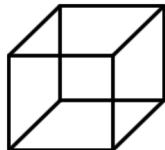


FIG. 2.1 – Le cube de Necker

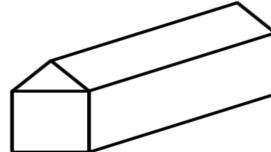


FIG. 2.2 – Quel(s) objet(s) représente ce dessin ?

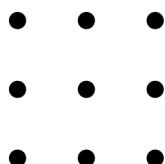


FIG. 2.3 – Passer par chacun des neufs points en traçant au plus quatre segments de droite sans lever le crayon

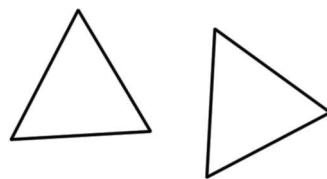


FIG. 2.4 – Former quatre triangles équilatéraux avec six allumettes

Le premier exemple est bien connu. Il fait la preuve que notre système visuel est sans cesse en train d'interpréter. Ceci est plus net encore dans l'exemple deux où l'on peut voir, en tournant éventuellement la feuille : a) une maison, b) une guérite, c) un lingot coupé en coin. L'exemple trois montre que notre système de compréhension peut à l'occasion, réduire arbitrairement l'espace de recherche : si l'on cherche une solution à l'intérieur du carré de neuf points, il est impossible d'en trouver ; mais l'énoncé n'impose nullement cette contrainte ; la solution (figure 2.5) est alors simple. Dans l'exemple quatre, deux contraintes superflues doivent être levées :

- a) les allumettes peuvent se croiser,
- b) la solution n'est pas nécessairement plane.

Il vient alors facilement deux solutions pour ce problème : l'une dans le plan, l'autre dans l'espace (figures 2.6 et 2.7).

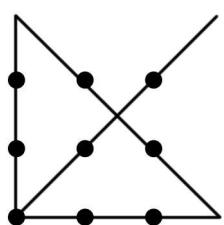


FIG. 2.5 – Exemple 3

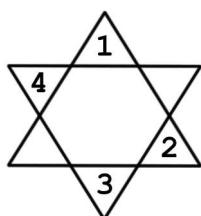


FIG. 2.6 – Solution dans le plan

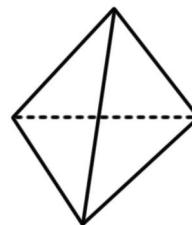


FIG. 2.7 – Solution dans l'espace

L'exemple suivant, quant à lui, ne fait intervenir que le langage : « Un roi cruel a fait mettre au cachot une jeune fille qui refusait de l'épouser. Après une année passée sans

que la jeune fille revienne sur sa décision, le roi la fait venir dans la cour du château et lui propose un marché :

Je vais ramasser deux cailloux, un noir et un blanc, lui dit-il, et les tenir cachés chacun dans une main. Tu choisisras alors librement l'une des deux. Si tu tires le caillou blanc tu seras libre ; si, au contraire, tu tires le noir tu m'épouseras.

La jeune fille accepte ce marché avec grande crainte. Mais sa crainte se transforme en panique quand elle voit que le roi se penche pour ramasser subrepticement deux cailloux noirs !

Que peut-elle faire ? »

Ce problème cité, notamment, par E. de Bono dans « La pensée latérale » illustre à nouveau une construction erronée de l'univers de recherche ; construction qui fait que l'énoncé obtenu n'a bel et bien aucune solution.

En réalité, la contrainte « Je vais ramasser un caillou blanc et un caillou noir », contrainte qui n'est pas respectée par le roi, se retourne joliment contre lui : la jeune fille se saisit du caillou de l'une des mains et le fait aussitôt tomber volontairement à terre au milieu des autres dans la cour.

« Pardon, dit-elle, mais la couleur de celui qui vous reste décide aussi bien de mon sort. »

L'autre est noir, bien sûr, et la jeune fille est libre.

Les exemples simples qui précédent montrent que les étapes 1 et 2 : « saisie de l'énoncé » et « inférences immédiates » sont généralement étroitement imbriquées. Il en va ainsi, en particulier lorsque le problème tel qu'il a été « compris » n'a pas de solution : c'est en cherchant une solution qu'on trouve alors le problème ; on fait alors abstraction d'une contrainte superflue.

Un cas différent, mais relativement fréquent, est celui où nous ne sommes capables d'aucune inférence immédiate : la résolution est de type : « tout ou rien », la solution jaillit seulement après un travail intérieur inconscient ; c'est l'effet appelé Hahal par Martin Gardner.

Soit par exemple le problème suivant :

« Vous vous trouvez seul, dans une pièce vide, avec deux barres de fer identiques, à ceci près que l'une est aimantée et l'autre pas. Il s'agit de déterminer celle qui est aimantée. (Les barres sont lourdes, solides, incassables et aucun autre matériel n'est à votre disposition.) »

Cet énoncé est en quelque sorte le seul de son espèce. Nous sommes démunis. Les opérateurs (les actions possibles) sont en nombre extrêmement réduit. Grande est la tentation de baisser les bras et de dire qu'il n'y a pas de solution ou bien que celle-ci est farfelue. Réfléchissez encore avant de lire la suite, **la solution existe et elle est simple.**

Un peu comme dans le problème précédent, il s'agit de rompre une symétrie. Mais ici la solution doit être purement physique et les seuls objets manipulables sont les deux barres. Or l'attraction magnétique est complètement symétrique par rapport aux deux barres, elle ne permet pas de dire qui attire et qui est attiré... sauf justement en un point : celui qui correspond au milieu de la barre aimantée, qui lui – par symétrie – ne peut être aimanté. On présente donc l'une des barres, A, perpendiculairement au milieu de l'autre B : si rien ne se passe, B est aimantée ; s'il y a attraction c'est B, non aimantée, qui est attirée vers A aimantée !

– Etape 3 : Jeu avec la situation

Cette étape est cruciale au début de la résolution. Montrons tout d'abord, toujours sur un exemple très simple, comment elle peut, à elle seule, donner l'idée de solutions élégantes.

Quoi de plus simple, pour démontrer le théorème de Pythagore que de jouer avec la règle et l'équerre ?

Une idée naturelle est de considérer l'équerre comme le triangle rectangle du théorème. Soient a et b ses côtés et c son hypothénuse. Pour faire apparaître $a + b$, il faut placer l'équerre le long de la règle de deux façons différentes. Un carré de côté $(a + b)$ peut être ainsi construit de deux manières (figure 2.8).



FIG. 2.8 – Deux carrés de côté $a + b$ pour le théorème de Pythagore

La surface S du grand carré est par construction $(a + b)^2$; or si T est la surface de l'équerre le dessin de gauche montre que S vaut : $4 * T + c^2$, tandis que celui de droite montre que S vaut aussi : $4 * T + a^2 + b^2$, ce qui démontre, après simplification par $4 * T$, le théorème de la manière annoncée.

2.6 Un exemple complet

L'exemple suivant reprend les points précédents pour aller jusqu'à un énoncé fermé et ensuite à la résolution du problème.

« Un damier de côté 100 est complètement rempli de 10000 pions noirs. Un coup légal, dans le jeu qui nous intéresse, consiste à changer sur une même rangée (ligne ou colonne) tous les pions noirs en pions blancs et tous les pions blancs en pions noirs. Est-il possible en un nombre fini de coups d'obtenir 1990 pions blancs sur le damier ? »

L'énoncé en français de ce problème est aisément compréhensible. Cependant :

- La réponse à la question posée ne va pas de soi ;
- Le « jeu avec la situation » est malaisé (les damiers 100×100 sont peu courants...).

La démarche naturelle, pour saisir l'énoncé et faire quelques inférences est, tout d'abord, de diminuer la complexité du problème en baissant la valeur des paramètres : on peut ainsi jouer, par exemple, sur un damier 4×4 , ou mieux, sur un carré « abstrait » de côté indéterminé.

La première déduction est sans doute alors la même pour tous : dès qu'un coup est joué, des pions blancs apparaissent. Ceci lève une bizarrerie éventuellement remarquée à la première lecture du texte : parler de pions blancs que l'on retourne a bien un sens

dès le deuxième coup joué. D'autres déductions viennent ensuite lorsque l'on joue sur le modèle de damier (l'ordre de ces déductions dépend des individus) :

- a) Si l'on joue deux fois de suite sur une même rangée, c'est comme si l'on n'avait rien fait.
- b) L'ordre des coups est indifférent : si je joue la ligne L_1 puis la ligne L_2 , c'est évidemment la même chose que si je joue L_2 puis L_1 , car ces deux lignes n'ont pas de pions communs ; il en va de même pour deux colonnes : s'il s'agit d'une ligne L_i et d'une colonne C_j . Elles n'ont en commun que le pion d'intersection qui sera, de toute façon, de sa couleur de départ si je joue d'abord j puis i et les autres pions ont bien la même couleur finale dans les deux cas. Notons ici que cette déduction – indifférence de l'ordre des coups – qui peut passer pour évidente chez l'adulte ne l'est pas du tout pour un enfant. Elle correspond au quatrième stade de développement de l'intelligence chez Piaget et n'est bien saisie qu'à partir de 10 ans environ.
- c) De plus, quel que soit le coup, le nom de la rangée est indifférent : ce n'est pas la place des pions noirs ou blancs qui compte dans le problème mais seulement leurs nombre.
- d) En vertu des deux premières déductions, on peut s'en tenir aux rangées touchées 1 ou 0 fois. En effet l'ordre étant indifférent, si une rangée a été touchée $2n + 1$ fois c'est comme si elle ne l'avait été qu'une fois, $2n$ fois revenant à 0 fois.
- e) En vertu de la troisième déduction, on peut alors regrouper les lignes modifiées 1 fois en haut du damier et les colonnes modifiées 1 fois à gauche du damier. Quatre rectangles recouvrent alors notre damier (cf. figure 2.9).

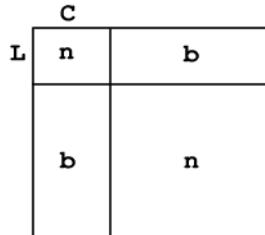


FIG. 2.9 – Le damier, les noirs et les blancs.

Les rectangles en haut à gauche et en bas à droite, sont remplis de pions noirs (les uns ont été retournés deux fois, les autres n'ont pas été touchés). Les rectangles en bas à gauche et en haut à droite sont remplis de pions blancs (car ils ont tous été retournés une fois).

- f) Seules comptent donc finalement les nombres L de lignes et C de colonnes qui ont été modifiées une fois.
- g) Le damier précédent possède une symétrie lignes/colonnes : si (L, C) est une solution au problème, alors (C, L) en est une autre par rotation de 90 degrés.
- h) Le damier précédent possède (c'est moins évident, vérifiez-le sur les équations infra. si vous n'êtes pas convaincu maintenant) une symétrie par rapport à son centre : si (L, C) est une solution alors $(100 - C, 100 - L)$ est aussi une solution. Remarque : Tout ce que nous venons de faire n'est pas réellement démontré. C'est même très difficilement formalisable puisque le point de départ est l'énoncé, ambigu et incomplet, en français, sur lequel s'appuient nos connaissances pragmatiques et notre système visuel. C'est pourtant – et cela pour tous les problèmes

– l'étape indispensable pour parvenir à un énoncé fermé où tout a été traduit et sur lequel un consensus peut s'établir.

Maintenant, poser cet énoncé est aisément résolu : seules interviennent les nombres entiers L et C (modulo 2) de rangées modifiées ; en décomptant les rectangles blancs, on a :

- en haut : $L \times (100 - C)$
- en bas : $C \times (100 - L)$ pions blancs

L'énoncé fermé obtenu, soit cf. 1, est l'équation en L et C :

$$100(L + C) - 2L \times C = 1990 \quad (2.1)$$

A ce stade, il est évident qu'on ne peut jamais obtenir un nombre impair de blancs. Mais, il est ici facile, pour avoir une meilleure vision globale de ce nouveau problème de le généraliser quelque peu : soit p le demi côté du damier et n la moitié du nombre cherché ; il vient après division par 2 :

$$p * (L + C) - L \times C = n \quad (2.2)$$

Maintenant, il est clair que :

- a) La difficulté du problème vient de ce que l'équation à résoudre est en nombres entiers ;
- b) La même équation rappelle quelques (vieux) souvenirs sur les coniques ($xy - ax - by - c = 0$) ;
- c) Ce qui est gênant, pour exhiber les solutions, est le mélange des termes linéaires : $p \times (L + C)$ et quadratiques : LC .

Les deux dernières inférences immédiates sur (2) se résument en l'argument suivant : une conique possède deux axes de symétries et son équation est bien plus agréable si elle est ramenée au centre de symétrie : autrement dit, il s'agit de faire disparaître le terme linéaire (gênant) par un changement de variables de la forme : $L = l + a$, $C = c + a$. Il vient :

$$\begin{aligned} p(l + c + 2a) - (l + a)(c + a) &= n \\ 2ap + p(l + c) - a(l + c) - lc - a^2 &= n \end{aligned}$$

et l'on prend $a = p$ pour ne garder que le terme quadratique (ceci n'est pas un hasard : on retrouve la symétrie par rapport au centre du damier, cf. dernière inférence immédiate n°8). L'énoncé fermé prend la nouvelle forme (3) :

$$lc = p^2 - n \quad (2.3)$$

avec $l = L - p$ et $c = C - p$.

Nous sommes ici dans un **troisième univers** : à gauche et à droite du signe $=$ se trouvent des entiers. La partie droite est connue ; elle vaut :

$$50 \times 50 - 1990/2 = 1505$$

Pour que le problème admette une solution, il faut que le nombre 1505 admette une **décomposition en facteurs premiers** compatible avec les dimensions du damier. Or, on a :

$$1505 = 5 \times 7 \times 43$$

$p = 50$ donc : $-50 \leq l \leq +50$ et $-50 \leq c \leq +50$.

Il existe bien deux solutions acceptables :

$l = 5 \times 7$ et $c = 43$ (ou bien sûr l'inverse)
 soit $L = 85$ et $C = 93$ d'une part. et :
 $l = -35$ et $c = -43$
 soit $L = 15$ et $C = 7$ d'autre part.

Il existe ainsi deux façons d'obtenir 1990 pions blancs.

Par contre, il existe des nombres de blancs impossibles à atteindre ainsi : $n = 1984 = 2500 - 992 = 2 \times 2 \times 13 \times 29$, et l'une des deux quantités l ou c dépasse 50, ce qui ne laisse place à aucune solution.

Enfin, le cas particulier où soit l , soit c est nul, c'est-à-dire L ou C égale 50 donne le cas dégénéré où quelle que soit la valeur de l'autre inconnue, le seul nombre qu'il est possible d'atteindre est $n = p^2$ soit $n = 2500$.

Tous les problèmes de la classe de départ peuvent être résolus par la même méthode, y compris ceux où le damier n'est plus carré mais rectangulaire. Quant aux autres méthodes pour résoudre l'énoncé général de type (1), c'est-à-dire celui des équations en nombres entiers, nous les retrouverons en particulier aux chapitres 3 et 6.

Le paragraphe suivant donne la démarche proposée par le mathématicien Georges Polya dans « How to solve it » (1956) pour résoudre un problème.

2.7 Pour résoudre un problème vous devez successivement

A — COMPRENDRE LE PROBLEME

- Quelle est l'inconnue ? Quelles sont les données ? Quelle est la condition ?
- Est-il possible de satisfaire à la condition ? La condition est-elle suffisante pour déterminer l'inconnue ? Est-elle insuffisante ? Redondante ? Contradictoire ?
- Dessinez une figure. Introduisez la notation appropriée.
- Distinguez les diverses parties de la condition. Pouvez-vous les formuler ?

B — CONCEVOIR UN PLAN

- Trouvez le rapport entre les données et l'inconnue.
- Vous pouvez être obligé de considérer des problèmes auxiliaires si vous ne trouvez pas un rapport immédiat.
- Vous devez obtenir finalement le plan de la solution.
- L'avez-vous déjà rencontré ? Ou bien avez-vous rencontré le même problème sous une forme légèrement différente ?
- Connaissez-vous un problème qui s'y rattache ? Connaissez-vous un théorème qui puisse être utile ?
- Regardez bien l'inconnue et essayez de penser à un problème qui vous soit familier et qui ait la même inconnue ou une inconnue similaire.
- Voici un problème qui se rattache au vôtre et que vous avez déjà résolu. Pourriez-vous servir de son résultat ? Pourriez-vous servir de sa méthode ? Vous faudrait-il introduire un élément auxiliaire quelconque pour vous en servir ?
- Pourriez-vous énoncer le problème différemment ? Pourriez-vous l'énoncer sous une autre forme encore ? Reportez-vous aux définitions.
- Si vous ne pouvez résoudre le problème qui vous est proposé, essayez de résoudre d'abord un problème qui s'y rattache. Pourriez-vous imaginer un pro-

blème qui s'y rattache et qui soit plus accessible ? Un problème plus général ? Un problème plus particulier ? Un problème analogue ? Pourriez-vous résoudre une partie du problème ? Ne gardez qu'une partie de la condition négligez l'autre partie ; dans quelle mesure l'inconnue est-elle alors déterminée, comment peut-elle varier ? Pourriez-vous tirer des données un élément utile ? Pourriez-vous penser à d'autres données qui pourraient vous permettre de déterminer l'inconnue ? Pourriez-vous changer l'inconnue, ou les données, ou toutes deux s'il est nécessaire, de façon que la nouvelle inconnue et les nouvelles données soient plus rapprochées les unes des autres ?

- Vous êtes-vous servi de toutes les données ? Vous êtes-vous servi de la condition toute entière ? Avez-vous tenu compte de toutes les notions essentielles que comportait le problème ?

C — METTRE UN PLAN A EXECUTION

- En mettant votre plan à exécution, vérifiez-en chaque détail l'un après l'autre. Pouvez-vous voir clairement si ce détail est correct ? Pouvez-vous démontrer qu'il est correct ?

D — EXAMINER LA SOLUTION OBTENUE

- Pouvez-vous vérifier le résultat ? Pouvez-vous vérifier le raisonnement ?
- Pouvez-vous obtenir le résultat différemment ? Pouvez-vous le voir d'un coup d'oeil ?
- Pouvez-vous vous servir du résultat ou de la méthode pour quelque autre problème ?

2.8 Petite histoire des mathématiques et de leur enseignement

L'histoire des mathématiques est précédée d'une longue préhistoire dont nous avons des traces remontant à quatre mille ans. Les animaux supérieurs, les jeunes enfants perçoivent dans notre monde deux entités abstraites fondamentales : le nombre et la forme. L'arithmétique et la géométrie furent ainsi, longtemps distinctes, les deux sciences fondamentales. Au départ la connaissance des nombres chez l'homme n'est pas très fine.

L'homme, dans les sociétés primitives, ne distingue pas deux ensembles presque égaux. Il sait à peine compter : un, deux, beaucoup. « Beaucoup » se dit « très » en latin ; ce mot subsiste encore aujourd'hui en français : « très » mais aussi « trois » !

2.8.1 Les notations au cours des âges

Des universaux sont repérés par les plus anciennes civilisations qui observent la ronde des astres dans le ciel. Nous savons ainsi que les Sumériens d'Uruk et de Nippur (-3000) utilisaient déjà un calendrier lunaire. Et l'idée leur vint de représenter les nombres par des symboles : la lune représente l'unité et des lunes accolées les nombres suivants. La nécessité de faire des comptes et de les écrire conduit à utiliser des abréviations plus commodes. La barre verticale ou oblique tient alors lieu d'unité (Phénicien, Syriaque, Nabatéen, Grec ancien, Sudarabique, Indien). Les ensembles de cinq, dix ou vingt unités sont abrégés par des symboles spéciaux éventuellement dérivés de leur nom.

Tous ces systèmes sont additifs, c'est-à-dire que le nombre codé est la somme des symboles représentés.

Les Babyloniens se distinguent en inventant le système sexagésimal : les symboles de base valent 1, 10, 60, puis 600, 3600, 36000 et ainsi de suite. Ce système s'est perpétué jusqu'à nous, par l'astronomie, pour les mesures sexagésimales de temps et d'angle. Plusieurs civilisations ont, de plus, l'idée d'utiliser les lettres de leur alphabet pour représenter également les nombres. Ceci permet de donner un sens à certains d'entre eux : ce sont les *calculs cabalistiques*. Le nombre correspondant à une lettre devient fonction de la **position** de celle-ci dans le mot ; la nécessité de marquer le « rien » se fait sentir.

L'origine du zéro reste toutefois obscure. Il existe de façon sûre dans des textes indiens du V^{ème} siècle où il prend la forme d'un point. Dans des écrits astronomiques grecs, le zéro est représenté par la lettre o initiale du mot grec *ονδεν* : « rien ».

La forme actuelle de nos chiffres, notre système décimal, vient de l'Inde de l'Ouest (forme gouzratie), par l'intermédiaire des Arabes. Mais ce n'est guère qu'au XVI^{ème} siècle qu'elle pénétra en Italie, adoptée par les commerçants de Florence. Son emploi n'est généralisé qu'au XVI^{ème} siècle. La marque de cette aventure subsiste d'ailleurs dans les mots « chiffre » et « zéro » qui viennent tous deux de l'arabe : sifr (zéro).

C'est l'invention de l'imprimerie (1440), qui fixe finalement la forme de ces dix symboles. L'usage de la virgule pour noter les nombres « réels » ne se répand qu'au XVII^{ème} siècle. Les quatre opérations sont connues des Egyptiens, mais leurs représentations sont souvent malcommodes : la juxtaposition marque l'addition et un ψ retourné marque, chez les Grecs, la soustraction. Finalement, ce sont les copistes du Moyen-Age qui abrègent puis déforment le mot « et », qui devient « + » tandis que l'habitude de séparer dans les comptes le poids de la tare à l'aide d'un tiret horizontal donne naissance au signe « - ». (signalons cependant que Marcel Cohen, dans son livre remarquable « La grande invention de l'écriture et son évolution » (Klincksieck, Paris 1958) donne + et – comme abréviations de « plus » et de « minus »). Les signes de multiplication et de division actuels ne sont introduits qu'au XVII^{ème} siècle. L'égalité est marquée en Europe au XVII^{ème} siècle par le symbole ∞ par lequel les astronomes désignent la constellation du Taureau. Mais le mot latin « aequalis » en toutes lettres se rencontre aussi, il est progressivement abrégé en æ et devient, finalement, le signe « = ». Le symbole ∞ désigne alors le nombre 1000.

C'est J. Wallis qui, vers 1660, l'élève au rang « d'infini »; auparavant, cette notion d'infini n'avait pas d'existence.

Pourquoi donc raconter tout cela, vous dites-vous ?

Pour deux raisons : la première est simplement de faire savoir que l'humanité a mis **plusieurs millénaires pour domestiquer le nombre** et que la science n'est ce qu'elle est que depuis quelques siècles. Les mathématiques ne se sont pas faites en un jour et, qui plus est, leur enfance n'est guère éloignée de nous. Quoi d'étonnant dès lors, puisque les hommes ont mis si longtemps à représenter les nombres et les opérations, à ce qu'un écolier rencontre à ce propos quelques difficultés ?

La seconde raison est que cette question de la **représentation des objets concrets ou abstraits est au cœur de l'intelligence artificielle**. Représenter quelque chose, cela signifie d'abord savoir l'isoler, avoir saisi son importance et son intérêt pratique. Cela implique ensuite que ses propriétés aient été repérées et traduites sous une forme manipulable. En un mot **représenter c'est comprendre**. Si la biologie commence aujourd'hui à expliquer la représentation de l'information héréditaire dans les gènes des

êtres vivants, les neurologistes et les psychologues sont encore loin de savoir expliquer comment sont codées et organisées nos connaissances dans notre cerveau. Il n'est pas exclu que les recherches en I.A. donnent ici des éléments de réponse. Mais continuons.

Depuis Euclide (-III^{ème} siècle), les géomètres savaient désigner des objets abstraits par des lettres. C'est Pappus d'Alexandrie (III^{ème} siècle), puis Diophante (IV^{ème} siècle), qui introduisirent progressivement des notations semblables pour des nombres inconnus. C'est François Viète, maître des requêtes sous Henri IV, qui dans son « *Ars analyticum* » fait, le premier, l'usage systématique de cette représentation littérale et est ainsi le précurseur de notre algèbre moderne. On trouve dans les écrits de Viète des expressions telles que :

$$\left[\begin{array}{c} H \ln D \\ -F \ln D \\ \hline F + D \end{array} \right] aequabitur A \quad (2.4)$$

Les premières lettres de l'alphabet, ici A et D, remplacent les inconnues ; le mot « *ln* » tient lieu de signe *, *la notation linéaire* d'aujourd'hui n'est pas acquise. Les deux expressions au-dessous du trait s'ajoutent ; ainsi, l'expression se lit-elle, en notations actuelles :

$$\frac{ay - by}{b + y} = x$$

Viète sait manipuler de telles expressions : de (4), il déduit : « *F + D* est à *H - F* comme *D* est à *A* », ce qui lui permet de résoudre des équations (1590).

Descartes au XVII^{ème}, normalise et répand ce symbolisme. C'est lui qui vulgarise nos « *+. -, *, /* » et l'usage des dernières lettres de l'alphabet pour désigner les inconnues. Le symbole « *x* » vient de l'arabe, comme figurant le son initial du mot « *sayc* » : une chose quelconque. Ainsi, c'est Descartes qui fait abandonner les signes cossiques, issus du grec et de l'hébreu, au profit des graphismes que nous connaissons aujourd'hui. Descartes réunit enfin la science des nombres et celle des figures en jetant les bases de la géométrie analytique et établit ainsi l'unité profonde des mathématiques. Celle-ci va être approfondie dans les siècles suivants.

Les indices et les exposants n'apparaissent que très tard : Euler (1707-1783) emploie encore couramment $x*x*x$ pour x^3 et c'est Evariste Galois (1811-1832) qui, le premier, fait usage d'indices. Mais Jordan, puis Hilbert lui-même, au début de ce siècle, les acceptent mal et continuent à écrire dans un style lourd et presque illisible où l'ordre des lettres dans l'alphabet tient lieu d'indexation implicite... Les symboles fonctionnels sont introduits par Leibnitz et Johann Bernoulli, le signe \sum par Euler.

La notation des fonctions ne va pas sans mal ; l'unanimité n'est pas faite aujourd'hui encore. On confond couramment la fonction f et sa valeur en un point : $f(x)$. On trouve encore pour la traduite par $+a$, la notation $T+a(f(x))$ au lieu de $T+a(f)$. La fonction dérivée est presque toujours confondue avec sa valeur, faute de symbolisme adéquat. Le cas des dérivées partielles est plus épineux encore. Et le problème se complique lorsque l'on veut considérer une expression comme fonction de l'un de ses constituants : on écrit, en mécanique ou en physique : $y(t)$, pour y considérée comme fonction du temps, puis $y(x)$ pour la fonction de la position et y tout court pour la fonction générale ! De telles notations sont inadmissibles car incohérentes et incompréhensibles du jeune étudiant. Il est, en revanche, impossible d'introduire un symbole pour chaque dépendance fonctionnelle.

Récemment (Church, Curry, 1950), a été proposée la λ -notation : les symboles λx , devant une expression, transforment cette dernière en une fonction de x : une fonction linéaire de x s'écrit ainsi $\lambda x(ax + b)$. Ceci résout donc très élégamment le problème. C'est à partir de cette λ -notation que fut défini, en 1960, par J. Mc Carthy, le langage de programmation LISP. La logique symbolique et les notations ensemblistes se développent par ailleurs très tard. C'est à Péano (1891) que nous devons le signe \in d'appartenance. L'inclusion est notée : α ou $<$, puis : \subset par Hausdorff (1920) : ce dernier signe prévaut aujourd'hui, mais il inclut ou non l'égalité suivant les auteurs et les pays. Après d'autres tentatives, ce sont à nouveau les symboles de Péano, \cap , \cup , \supset qui prévalent pour l'intersection, l'union et l'implication logique. Comme ce dernier symbole correspond au signe renversé de l'inclusion et que fâcheusement : si $E \subset F$ alors $(x \in E) \supset (x \in F)$, Hilbert se sert de la flèche \rightarrow pour marquer cette implication.

Mais elle se confond elle-même avec le signe d'application ensembliste et celui de réécriture. L'école Bourbaki préfère le signe \Rightarrow qui est d'usage courant aujourd'hui. Nous utiliserons, quant à nous, le symbole de Péano qui reste celui des logiciens.

Afin de mettre en forme les raisonnements logiques, c'est Euler (1707-1783) qui eut l'idée des diagrammes ressemblant à des cercles pour figurer les ensembles et qu'on appelle aujourd'hui... diagrammes de Venn (1834-1923). Lewis Carroll (1832-1898) proposait, quant à lui, un autre type de diagrammes qui conservait la dualité entre un ensemble et son complémentaire : il est utilisé aujourd'hui en calcul booléen.

Par influence des symboles \cup et \cap , le « ou » et le « et » logiques sont notés respectivement \vee et \wedge .

La négation est représentée tantôt par le signe \neg , tantôt par une barre horizontale, tantôt par le signe \sim . Comme ils faisaient tous double emploi, A. Heyting propose, en 1937, le signe \neg que nous adopterons. Le quantificateur existentiel \exists est lui aussi dû à Péano. Russel et Whitehead (1903) le complètent par la quantification universelle qu'ils indiquent simplement en marquant la variable quantifiée entre parenthèses. Ce n'est que plus tard (1920) que le signe \forall fut ajouté explicitement. C'est à G. Frege (1940) que revient l'idée de la nécessité d'un signe d'affirmation qu'il note \vdash

$\vdash A$ signifie : « on a A » .

L'école mathématicienne française de renom qui a publié, sous le pseudonyme de Nicolas Bourbaki les monographies « Eléments de mathématique » a fixé l'usage de bon nombre de ces symboles et en a introduit de nouveaux : outre le signe \Rightarrow déjà mentionné, la notation C pour le complément d'un ensemble et de nombreux signes en théorie des groupes. Par ses efforts de clarté, de simplification et de rigueur, ainsi que par les nombreuses traductions qu'elle a connue, cette œuvre a fait beaucoup pour figer les notations, l'esprit et le style mathématique qui ont cours aujourd'hui. Mais une différence importante s'est progressivement créée entre la syntaxe du langage naturel et celle des langages formels. Là où le langage met par exemple des séparateurs – « et », « ou », « car » – le mathématicien a préféré des structures agrégeantes comme les parenthèses, crochets, accolades et barres horizontales. Le symbole d'égalité, qui signifie l'identité des deux objets, reste, quant à lui, un séparateur privilégié. Il en résulte une certaine confusion entre les deux langages et il est courant de lire des formules mathématiques de style naïf comme :

« pour tout $y = ax + b$ »

dans lequel le symbole y sert deux fois (on veut écrire en réalité : $\forall y, y = ax + b$: alors que la formule donnée se lit : quelle que soit l'égalité du type $y = ax + b$).

En fait, il y a confusion entre la vraie signification du signe $=$, donnée ci-dessus, et sa valeur opératoire. Ainsi les « identités remarquables », telle $(a - b)^2 = a^2 - 2ab + b^2$ sont d'abord des règles de réécriture aussi bien que $x + o \rightarrow o$ ou $x \cdot 1 \rightarrow x$. Le signe $=$ sert aussi à exprimer des définitions, comme dans $\tan x = \sin x / \cos x$: ou encore à introduire des abréviations :

Je pose $u = (a - x + b)/(x^2 + 1)$, ou : $x', x \Rightarrow (-b \pm \sqrt{\Delta})/2$ dans lesquels le membre droit est toujours utilisé **en bloc**.

D'autres ambiguïtés nées du langage courant se propagent en mathématiques : quand on définit un groupe G , on postule d'abord pour tous les g de G , l'existence d'un e tel que $ge = g$ et $eg = g$. On poursuit alors avec :

$$\forall g, g \in G, \exists g^{-1} \in G \text{ et } g^{-1}g = e, gg^{-1} = e$$

Il y a là un bel abus de langage mathématique puisque la variable g quantifiée existentiellement dans la première phrase n'a a priori aucun rapport avec celle de la seconde. Ce genre d'abus ne va pas sans causer quelques difficultés à ceux qui écrivent des démonstrateurs automatiques de théorèmes.

La simple traduction des phrases françaises en langage mathématique se doit de traiter avec soin les quantificateurs. « Un triangle rectangle est un triangle qui a un angle droit »... Le premier « un » est universel, le troisième est existentiel...

Le verbe « être », quant à lui, est tantôt la marque d'une définition, tantôt celle d'une propriété, tantôt celle de l'appartenance.

Le « ou » tantôt exclusif « blanc **ou** noir », tantôt inclusif « peu **ou** prou » correspond à deux notions mathématiques différentes. Très tôt on enseigne simultanément aux enfants des phrases pseudo-mathématiques comme : « 2 par 2 donne 4 » et « plus par moins donne moins ». Ce type d'assimilation osée et non expliquée crée souvent quelques blocages. Ceux-ci finissent par ressortir de façon inattendue :

« Quel entier y a-t-il après n ? » demande le professeur ; « o » répond l'élève.

« Soit l'entier q ... », « mais, interrompt l'élève, q n'est pas un entier, c'est une lettre »...

Et quelle doit être la réponse à l'exercice : « Donner l'ensemble des x de l'alphabet, tel que x est une voyelle » ? L'ensemble vide, car x est une consonne ?...

Ainsi, le **passage d'un énoncé en français à un énoncé en langage formalisé** n'est pas chose aisée ; c'est la première étape dans toutes les disciplines scientifiques : algèbre, géométrie, analyse, chimie, physique. Les chercheurs opérationnels qui traduisent souvent des situations complexes lui ont donné le nom de *modélisation*.

Par ailleurs, le langage mathématique n'est pas exempt de convention et il pêche souvent par omission. La signification des symboles est le plus souvent contenue implicitement dans les symboles eux-mêmes : les « inconnues » se notent x, y, z ; les entiers i, j, k, l, m, n, p, q ; les réels r, s, t , les angles $\alpha, \beta, \theta, \phi$; les fonctions f, g, h ; alors que les ensembles sont repérés par des lettres majuscules. Certaines lettres : e, g, i et π désignent des valeurs réservées. Le langage mathématique reprend, en outre, les mauvaises habitudes du langage naturel ; ainsi un « groupe-de-Lie » n'est pas plus un groupe qu'un « pied-à-terre » n'est un pied. Dès lors, l'étudiant doit savoir **lire entre les lignes** et comprendre à demi-mot.

Il est temps de s'intéresser à cet étudiant, à ce qu'on lui demande, à l'histoire de l'enseignement de ces mathématiques.

2.8.2 L'enseignement

Les « écoles » grecques de Socrate (- V^{ème} siècle), de Platon (- IV^{ème} siècle), d'Aristote (- IV^{ème} siècle), d'Euclide (- III^{ème} siècle) sont justement célèbres. Les « Eléments de géométrie » de ce dernier sont restés longtemps un modèle de rigueur et de clarté. Vers 290, Pappus d'Alexandrie publie sa « Collection mathématique » et vers 370, Diophante son « Arithmétique ». Après ces développements originaux et importants des Grecs, l'Europe connaît une longue période de stagnation dans les sciences exactes. Il faut attendre la Renaissance Italienne pour donner un élan nouveau tant aux recherches qu'à l'enseignement.

Les mathématiques sont encore jusque là réservées à une élite : on ne les pratique que dans les cercles philosophiques ou les salons. C'est d'ailleurs le maître de philosophie qui les professe dans les collèges ; quelques écoles spécialisées : architecture, astronomie, navigation, art militaire, développent par ailleurs les techniques de calcul.

La Révolution française, avec la généralisation de l'école, va donner à l'enseignement des sciences exactes le statut que nous lui connaissons (elle impose en outre l'usage simplificateur du système métrique). En 1794, Lakanal et Monge créent l'Ecole Polytechnique. En 1808, la chaire de « Mathématiques transcendantes » est créée à la Sorbonne, qui, depuis sa naissance (Robert de Sorbon 1227), était un collège d'études théologiques. Depuis lors, les programmes d'enseignement des lycées et collèges sont calqués sur ceux des concours d'entrée à ces écoles.

On enseigne donc aux élèves **les mathématiques du programme** et non à faire des mathématiques. Remarquablement peu nombreux sont les auteurs qui s'intéressent au phénomène de la découverte, aux méthodes « actives » d'enseignement, au raisonnement inductif, à l'étude des cheminements de la pensée, à *l'heuristique* en pédagogie (du verbe grec *ενρεκειν* « trouver », il s'agit de faire découvrir par l'élève lui-même ce qu'on veut lui apprendre, à la *διδασκειν* des disciplines, du verbe grec *didaskein* « enseigner ». Il s'agit d'étudier pour lui-même le phénomène enseignement dans ses aspects cognitifs et techniques et de l'évaluer dans toutes ses dimensions).

Célestin Freinet crée en 1940 une école nouvelle qui veut éviter le côté illusoire du discours pédagogique habituel, et prône la participation et le travail en groupe.

Nous ne possédons que de très rares textes où un mathématicien explique comment il est réellement parvenu à un résultat important. On peut même dire que dans la rédaction des textes en maths, tout est fait pour cacher la démarche inductive d'origine. Le raisonnement est, la plupart du temps, présenté de façon inversée par rapport à son invention. La preuve est épurée des arguments heuristiques. Les représentations sont changées et codées. Enfin, les objets introduits sont parachutés ! Les enseignants ont aujourd'hui parfaitement conscience de ces problèmes, mais les réponses ne sont pas simples. L'introduction des mathématiques « modernes » fut une tentative - mal reçue - pour résoudre certains d'entre eux. En intelligence artificielle, c'est précisément cette démarche heuristique qui intéresse les chercheurs dans tous les domaines d'application.

Šuruppak, vers - 2650. — Extrait de R. JESTIN, *Tablettes sumériennes de Šuruppak*, Paris, 1937.
n° 50, face, revers non inscrit — Dimension : 0 m. 080 × 0 m. 083.



I.	<i>i gur, it</i>	[étant donné] 1 grenier d'orge :
II.	<i>7 silā</i>	7 [mesures dites] silā
III.	<i>lī-ā iu-ha-ti</i>	chaque homme reçoit.
IV.	<i>lī-bi</i>	ces hommes [sont] :
V.	4 (fois) 36.000	144.000 + 18.000 + 2.400
	+ 5 —	+ 120 + 51 = 164.571.
	+ 4 —	600
	+ 2 —	60
	+ 5 —	10
	+ 1	
VI.	3 silā <i>it lu-kid</i>	il reste 3 silā d'orge.

Note. — 1 grenier d'orge vaut 1.152.000 silā.

Interprétation de Maurice Lambert

2.9 Les représentations

Notre cerveau travaille non pas sur la réalité mais sur un **modèle** de cette réalité. Ce modèle lui est transmis par nos capteurs : systèmes visuel, auditif, olfactif et gustatif, toucher. Le modèle est ensuite **représenté** dans le système nerveux à travers les neurones et les synapses. Le mécanisme physiologique qui code, mémorise et accède alors à l'information, nous est à peu près complètement inconnu. Selon Konrad Lorenz, cette représentation est essentiellement une **image visuelle du réel** : les singes supérieurs ont acquis l'intelligence parce qu'ils furent contraints, pour survivre dans un univers hostile, de « prévoir » leurs actions et leurs comportements, de « réfléchir ». Les traces dans le langage courant de cette prédominance des rapports spatiaux dans nos schémas de pensée sont nombreuses : voir, saisir, comprendre ; niveau, plan, profondeur ; évident, obscur. C'est l'**espace** qui sert de repère pour tous les rapports abstraits. Et pourtant, c'est là une caractéristique bien humaine, ces rapports abstraits ne sont pas innés. L'enfant les forge petit à petit, ne saisissant les invariants physiques qu'après un long apprentissage. Selon Piaget, la conservation du poids ou du volume des objets n'est assimilée, chez l'homme, qu'à 13-14 ans. Il n'empêche que, petit à petit, cette représentation abstraite du monde conduit l'individu à la **pensée pure** indépendante de tout support et séparée de toute action immédiate.

Finalement, les représentations, que nous utilisons, qu'il s'agisse de textes écrits, de gravures, de compositions musicales, d'énoncés mathématiques, de dessins, de photos, sont essentiellement **graphiques**. Parmi celles-ci, les notations mathématiques jouent, dans cet ouvrage, un rôle particulier.

2.9.1 Les notations mathématiques

Tous les systèmes habituels de notations comportent, d'une part, des symboles représentant des *objets* et, d'autre part, des *opérateurs* permettant de relier les objets entre eux. Ainsi, dans l'expression : $x + y - 1$, x , y et 1 sont des objets et les signes $+$ et $-$

sont des opérateurs.

Un opérateur relie en fait un nombre bien déterminé d'objets : comme $+$ et $-$, les symboles $*$, $/$, \cup , \cap , \vee , \wedge , \supset relient deux objets et sont dits binaires ; $\sqrt{}$, Log , $| \cdot |$, sinus, sont des opérateurs unaires : le \sum est le symbole ternaire (variable liée, ensemble de variation, terme général), de même que le if... then... else... (si... alors... sinon...), des langages de programmation. Un opérateur n -aire suivi de ses n objets est *une expression bien formée ou terme*.

Si un opérateur est n -aire, on dit aussi que son *poids* est n .

Une expression bien formée peut, elle-même, être considérée comme un nouvel objet ; elle peut, à son tour, être reliée à d'autres par un opérateur suivant la même loi de formation et, finalement un terme est un opérateur n -aire suivi de n termes.

Cette définition, qui définit un terme à partir d'autres termes, est dite *récursive*. Elle n'a de sens que parce qu'il existe bien, a priori, des termes de départ qui sont les objets.

Parmi ces objets de départ, certains peuvent, en général, être remplacés par d'autres et plus généralement par des termes quelconques, ce sont les *variables* et d'autres ne le peuvent pas, ce sont les *constantes*. Ainsi, en trigonométrie, dans l'expression : $\sin^2 x + \cos^2 x = 1$, x peut être remplacé par n'importe quel terme, mais pas les symboles 1 et 2, ni les symboles sin et cos.

Tout ceci sera précisé au chapitre 3 sur les **systèmes formels**.

2.9.2 Les notations linéaires

L'imprimerie a conduit le mathématicien à écrire ses formes sur des **lignes** lues, dans le monde occidental, de gauche à droite.

Ceci étant, trois possibilités subsistent encore pour ces notations dites linéaires : l'opérateur peut figurer, soit avant, soit après les termes qu'il commande ou bien encore au milieu d'eux. En fait, ces trois notations sont utilisées, mais de manières inégales.

La notation la plus commune est dite *infixée*, c'est celle où les opérateurs sont noyés au milieu des termes. Pour faciliter la lecture, des *parenthèses* : (\cdot) , $[\cdot]$, $\{\cdot\}$, **qui n'appartiennent pas** au symbolisme de base, ont dû être ajoutées. Elles sont lourdes et pénibles à écrire, aussi ne les écrit-on que pour lever les ambiguïtés ou bien pour scinder des expressions trop longues. En outre, on convient d'une hiérarchie entre les opérateurs. Ainsi, en algèbre, l'expression :

$a - b + c$ sera interprétée comme $(a - b) + c$ et non comme $a - (b + c)$. L'associativité et la commutativité de certains opérateurs ($+$ et $*$ surtout) sont implicites en analyse habituelle : on écrit $2 * x * y$ pour $x * 2 * y$ ou $y * x * 2$ et, le plus souvent, l'opérateur $*$ est lui-même omis. En réalité, la fantaisie va encore plus loin puisque les opérateurs unaires, par exemple, sont notés indifféremment sur trois niveaux et maintes variantes : $\sin A$, B , C^t , dD/dt , E^* , F , G' , $|G|$... Le signe $-$ est à la fois unaire et binaire... La notation infixée usuelle est donc loin d'être propre et rigoureuse.

Les logiciens, quant à eux, utilisent de préférence une notation plus pure, dite *préfixée*. Chaque opérateur est systématiquement écrit **avant** les termes qu'il commande :

' x opérateur y ' devient : 'opérateur x y '
tandis que : 'opérateur z ' est inchangé.

Tout système de parenthèses est alors inutile et cette notation, plus concise, qui place en outre l'opérateur principal en tête de chaque terme, devient, avec quelque habitude, bien agréable.

L'expression $(p \supset q) \supset p$ s'écrit en notation infixée :

$$\supset\supset p \ q \ p$$

L'expression (T) suivante : $\log(y + \sqrt{y^2 - b/\sin x})$ s'écrit quant à elle :

$$\log + y \ \sqrt{- \uparrow y \ 2 / b \sin x}$$

Théorème fondamental de la notation préfixée (Rosembloom).

Une suite s de symboles est une expression bien formée en notation polonaise préfixée si. et seulement si :

- a) rang (s) = -1
- b) rang (sous-suite gauche de s) ≥ 0

$$\left[\begin{array}{l} \text{avec rang (opérateur) = poids (opérateur) -1 soit :} \\ \text{rang (suite vide) = 0,} \\ \text{rang (symbole n-aire) = } n - 1, \\ \text{rang (variable) = rang (constante) = -1,} \\ \text{rang (s}_1 \text{ concaténée à s}_2 \text{) = rang (s}_1 \text{) + rang (s}_2 \text{).} \end{array} \right]$$

La démonstration, très naturelle, de ce théorème se fait par induction sur le nombre de symboles de s . Un corollaire important de ce résultat est de fournir un algorithme pour trouver dans une expression l'endroit où se termine un terme qui commence en une position donnée.

Ainsi, avec la suite de symboles :

$$\log + y \ \sqrt{- \uparrow y \ 2 / b \ sin \ x}$$

on a bien :

$$\begin{matrix} \text{rang}(s) & 0 & 1 & 0 & 0 & 1 & 2 & 1 & 0 & 1 & 0 & 0 & -1 \end{matrix}$$

et le terme qui commence par le signe \uparrow

$$\begin{matrix} 1 & 0 & -1 \end{matrix}$$

finit au signe 2.

Remarques :

- 1) Tout se passe comme si les variables et constantes étaient des opérateurs de rang -1.
- 2) La notation infixée, si elle est complètement parenthésée, peut être considérée comme préfixée et justifiable de la vérification précédente : toute parenthèse ouvrante est comptée comme un opérateur binaire et toute fermante comme une constante ; les autres rangs sont inchangés. On a en effet :

$$\begin{matrix} (& \text{a opérateur-binaire} & b &) \\ 1 & 0 & 1 & 0 & -1 \end{matrix}$$

Dans la notation *suffixée* l'opérateur est placé juste derrière ses termes et :

' x opérateur y ' s'écrit : ' $x \ y$ opérateur'

'opérateur z ' s'écrit : ' z opérateur'

L'expression (T) devient ici :

$$y \ y \ 2 \ \uparrow b \ x \ \sin / - \ \sqrt + \ \log$$

Cette notation est en particulier utile lorsque l'on désire **évaluer** l'expression représentée ; c'est le cas des calculatrices électroniques, mais aussi celui des compilateurs de langages de programmation qui traduisent en **postfixée** les formules entrées en infixées. Dans cette notation, en effet, un parcours de l'expression de gauche à droite permet le calcul de sa valeur en un seul passage.

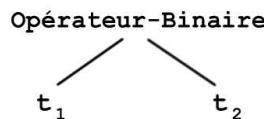
Exemple : Si $x = \frac{\pi}{2}$, $b = 80$, $y = 1$, la formule (T) est ainsi évaluée :

y	y	2	↑	b	x	sin	/	-	✓	+	log
1	1	2	↑								
				1	80	$-\pi/2$					
						$\rightarrow -1$					
							$\rightarrow -80$				
								$\rightarrow 81$			
									$\rightarrow 9$		
										$\rightarrow 10$	
											$\rightarrow = 1$

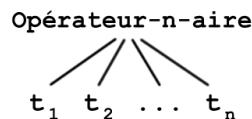
2.9.3 Notations non linéaires

La typographie, qui a induit la linéarité, impose des contraintes qui, visuellement, ne reflètent pas la vraie structure des expressions. Elle met en effet tous les symboles sur un plan d'égalité alors que nous savons que certains d'entre eux sont terminaux (variables et constantes) et les autres pas.

De façon précise, chaque opérateur *contrôle* un nombre donné de termes connus. Or ceci peut être exactement représenté sous forme de **dessins** du type :



et de façon générale :



pour un terme comportant un opérateur et ses n sous-termes.

De tels dessins sont appelés **arbres**.

Ceci correspond bien à notre vision mentale d'une expression telle que (T) dont le dessin serait semblable à la figure 2.10 :

Il est d'ailleurs remarquable que **cette notation contient les trois autres** :

- on retrouve la **préfixée** par un parcours de haut en bas et de gauche à droite de cet arbre, en prenant chaque symbole dès qu'il est rencontré ;
- on retrouve la **postfixée** à l'aide du même parcours, mais en ne prenant un symbole que lors du dernier passage ;
- on retrouve l'**infixée** parenthésée par une projection sur un axe horizontal en ajoutant les parenthèses convenables par niveau.

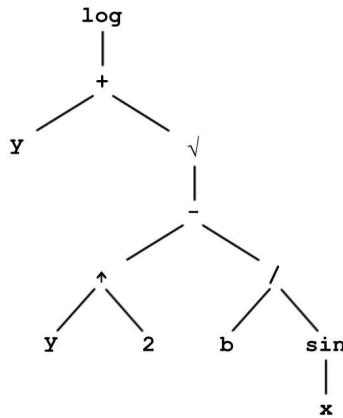


FIG. 2.10 – Représentation d'une expression en arbre

De plus, comme nous l'avons déjà souligné, une **représentation ne vaut que par la facilité de mise en œuvre des procédures de traitement**. Or, en mathématiques, les procédures courantes sont : **substituer** un terme à un autre, **regrouper** deux termes, **ôter** un terme. A toutes ces opérations correspondent des traitements très simples sur l'arbre : déplacement d'une *branche*, modification locale de symboles terminaux – *les feuilles* –, suppression d'une branche. Au lieu de cela, dans la notation infixée habituelle, nous réécrivons sans cesse à la main, ligne après ligne, l'expression toute entière.

2.9.4 Représentation en machine des expressions

La mémoire des ordinateurs est de grande taille et rapidement utilisable : nous pouvons nous affranchir de nos propres contraintes en mettant en machine de telles structures d'arbres. A chaque symbole d'une expression faisons correspondre trois informations : le nom de symbole lui-même, le « fils » gauche et le « frère » droit. Les fils sont les premiers symboles des termes correspondants. L'ensemble des noms de symboles sera réuni dans un tableau à un indice de nom S , celui des fils dans un tableau similaire de nom G , enfin les frères dans un tableau similaire D . Les trois tableaux S , G , D contiendront simultanément, pour un programme donné, l'ensemble des expressions qu'il doit manipuler et pourront donc être de grandes tailles.

L'idée de la représentation en machine est alors de faire correspondre à :

Opérateur 	quelque chose comme :	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 33.33%;">S</th> <th style="width: 33.33%;">G</th> <th style="width: 33.33%;">D</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">opérateur</td> <td style="text-align: center;">a</td> <td style="text-align: center;">b</td> </tr> </tbody> </table>	S	G	D	opérateur	a	b
S	G	D						
opérateur	a	b						

Mais, puisqu'en réalité, a et b peuvent être eux-mêmes des arbres, les informations en G et en O ne seront non pas a et b eux-mêmes, mais des *indices* dans le tableau S des endroits où l'on trouve a et b et donc les termes qu'ils commandent . Si les informations G et O aux niveaux a et b ne sont pas vides, elles permettront, à leur tour, de poursuivre le parcours de l'arbre.

Les extrémités terminales ou *feuilles* qui sont les variables et constantes n'ont pas de fils (G égal au symbole vide 0).

Ainsi, pour un indice i donné on a :

- $S(i)$ = nom du symbole
- $G(i)$ = indice du fils gauche de $S(i)$ dans le tableau S
- $O(i)$ = indice du frère droit de $S(i)$ dans le tableau S .

Avec cette notation, rien n'oblige à ce que les symboles consécutifs d'une expression se retrouvent dans l'ordre des indices croissants, puisque seul importe le parcours de l'expression, donné par les tableaux G et D .

L'ensemble des tableaux S , G , D constitue une *structure de liste*. Il est couramment utilisé pour la manipulation symbolique sur ordinateur.

L'expression précédente $\log(y + \sqrt{y^2 - b/\sin x})$ sera par exemple mise sous-forme de la liste suivante :

112	log	113	0
113	+	119	0

119	y	0	121
120	*	140	218
121	√	135	0

135	-	136	0
136	↑	255	137
137	/	138	0
138	b	0	139
139	sin	140	0
140	x	0	0

255	y	0	256
256	2	0	0

431	t	433	0
432	-	431	0
433	/	434	0
434	π	0	435
435	4	0	0

FIG. 2.11 – Représentation d'une expression en liste

Remarques :

- 1) L'expression elle-même est repérée par sa *tête de liste* c'est-à-dire l'indice de son premier symbole dans le système *S, G, D*. L'expression (T) est donc repérée par 112.
- 2) Le plus souvent, le tableau *S* contient en fait lui-même non pas le nom du symbole mais, à nouveau, un indice qui renvoie à *un dictionnaire*.
- 3) Si l'on doit traiter des opérateurs d'ordre supérieur à deux, il est ainsi possible de représenter des opérateurs d'ordre variable, comme le + généralisé : + x y z t tenant lieu de

$$x + y + z + t$$

- 4) Dans cette représentation l'opération fondamentale (cf. chapitre 3) de **substitution d'une variable par un terme** est particulièrement aisée. Ainsi, supposons que, dans l'expression précédente, je veuille changer *x* en (*t* - $\pi/4$), ce dernier terme figurant déjà quelque part dans la liste, par exemple à l'indice 432. Il suffit alors, pour effectuer la substitution prévue, de changer le pointeur gauche de *x*, soit *G(139)* et de faire *G(139) = 432*. L'enregistrement 140 esl alors *mort* puisqu'il n'est atteint par aucun point *G* ou *D*.
- 5) Il n'est pas interdit que plusieurs pointeurs atteignent le même enregistrement. **Cela n'est pourtant pas conseillé** : si *y* doit être à son tour changé en (*t* - $\pi/4$), Il est préférable de recopier un deuxième exemplaire de ce terme pour effectuer les deux substitutions sur *y* dans (T), car rien n'indique que les traitements ultérieurs sur *x* et *y* substitués seront identiques.

-
- 6) Il y a lieu lorsque les listes deviennent envahissantes de récupérer les noeuds morts pour pouvoir y mettre de nouveaux triplets. Ceci se fait à l'aide d'un programme *ramasse-miettes* (garbage collector)

2.10 Les modélisation graphiques en Intelligence Artificielle

En intelligence artificielle, on s'intéresse au traitement automatique de toute situation pour laquelle on ne dispose pas d'algorithme. Nous étudierons ici en particulier trois grands domaines d'application de l'intelligence artificielle : a) La démonstration de théorèmes par ordinateur ; b) La résolution automatique de problèmes ; c) La compréhension du langage naturel. A l'intérieur de ces trois domaines, nous nous intéressons essentiellement à la façon dont sont représentées et utilisées les connaissances. Si nos comportements méritent le qualificatif « d'intelligents », c'est bien, sans doute, en effet, parce qu'ils **mettent en jeu des représentations adaptées**, des schémas unificateurs, des abstractions raisonnables de notre univers.

2.10.1 Représentation des connaissances en démonstration automatique :

En mathématique comme dans les autres domaines, nous rencontrerons en fait deux niveaux de représentation : *Niveau 1* : La représentation des **objets** manipulés (êtres mathématiques, formules, expressions). *Niveau 2* : La représentation de la **connaissance** sur ces objets (relations, théorèmes, méthodes de démonstration).

A) Représentation des objets manipulés : La représentation des figures joue encore, pour nous, un rôle fondamental en géométrie : des programmes, qui démontrent des théorèmes, trouvent des lieux, font des constructions à la règle et au compas en géométrie plane, utilisent également un graphe pour représenter la figure en machine.

B) Représentation de la connaissance : Ce que les livres ne disent pas, ce que peu de professeurs de mathématiques enseignent, c'est **comment** on trouve une démonstration. En fait, la démarche du mathématicien comporte trois phases distinctes (voir Pastre, Polya, Mérialdo) : 1) **Comprendre** l'énoncé formel du théorème à démontrer i.e. le traduire dans la représentation interne du mathématicien ; 2) **démontrer** le théorème dans cette représentation en utilisant toutes les connaissances que le mathématicien possède et mémorise, elles aussi, dans cette représentation adaptée : 3) **traduire** la preuve incomplète en démonstration rigoureuse, en réintroduisant la formalisation symbolique du langage mathématique habituel (voir figure 2.12).

Le passage direct de I à IV est très difficile, les idées directrices utilisées dans (2) ayant disparu. Bien sûr, nous n'avons que difficilement accès aux représentations utilisées par les mathématiciens. Cependant certaines études ont été menées et des schémas équivalents à ceux des mathématiciens ont déjà programmés.

Avant d'en donner quelques exemples, mettons clairement en évidence le rôle joué en démonstration de théorèmes par « un bon dessin ».

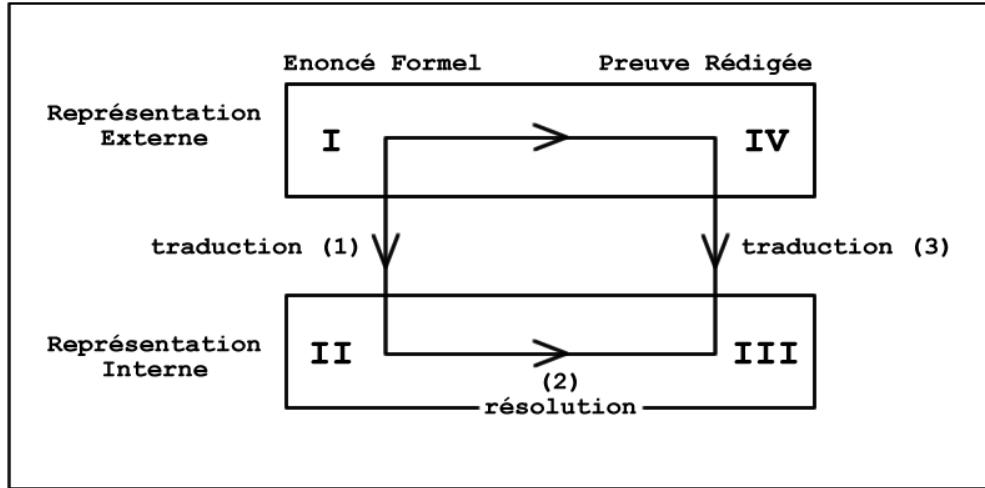
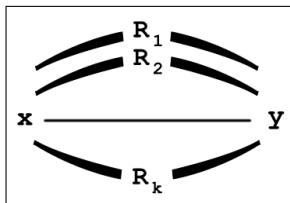


FIG. 2.12 – Le rôle de la représentation en mathématique

C) Le rôle de la figure : La figure est d'abord un abrégé de l'information, c'est une mémorisation commode des éléments essentiels qui interviennent. Ainsi deux éléments x et y peuvent être liés dans l'énoncé par plusieurs relations binaires R_1, R_2, \dots, R_k . Le schéma :



résume alors avantageusement l'écriture :

$$x R_1 y \wedge x R_2 y \wedge \dots \wedge x R_k y \text{ (Voir Bundl, Pastre)}$$

La figure, en liaison avec des algorithmes simples de traitement, peut traduire efficacement une chaîne de déductions. Pour une relation binaire R transitive, comme $=$ ou \subset ou \in , les schémas de type précédent permettent de traduire :

$$\begin{aligned} x R y \wedge y R z \text{ sous la forme : } & x \xrightarrow{R} y \xrightarrow{R} z \\ \text{et d'en déduire directement : } & x \xrightarrow{R} z \end{aligned}$$

Les étapes « évidentes » correspondantes peuvent alors être sautées pour obtenir une démonstration « en raccourci ».

La figure permet *d'éliminer des sous-but faux*. Si une propriété n'est pas vérifiée sur la figure, elle est nécessairement fausse et il est inutile de perdre du temps à s'efforcer de la démontrer. Inversement :

La figure **sert d'aide à la découverte**. Si une propriété est vérifiée sur la figure, alors on peut envisager de démontrer le théorème correspondant (Nous avons tous utilisé ce procédé pour trouver des lieux en géométrie).

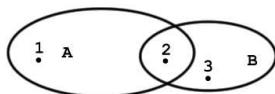
La figure est utile dans *l'étude des symétries des variables* et aide au repérage direct des situations équivalentes.

La figure permet enfin d'introduire de bons éléments nouveaux quand cela est utile à la démonstration (voir l'utilisation de ceci en géométrie).

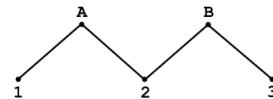
En résumé, la figure et les algorithmes de traitement qui lui sont liés, rendent compte de la structure utile des concepts concernés, c'est-à-dire leur sémantique, alors que l'énumération de leurs propriétés formelles, qui est la seule à apparaître dans les livres, ne décrit que leur syntaxe.

D) Exemples : Représentation des ensembles en topologie (B. Mérialdo) :

Le mathématicien utilise les diagrammes de Venn :



qu'il est facile de convertir en graphes :



Les sommets numérotés sont appelés atomes ; chaque atome est le représentant du sous-ensemble auquel il appartient. Si un atome est vide, il est ôté du graphe. Ainsi, les relations d'appartenance, d'inclusion, d'égalité sont mémorisées très concisément. En outre, comme pour les diagrammes de Venn, il est facile de décrire les algorithmes simples, qui, sur cette représentation, permettent d'ajouter un ensemble, de construire une intersection, d'imposer une égalité, de regarder si une inclusion est vérifiée, ou de déterminer le complémentaire d'un ensemble. Ainsi B est inclus dans A si, et seulement si, pour toute arête reliant le sommet B à l'atome I, il existe une arête liant également A à I :

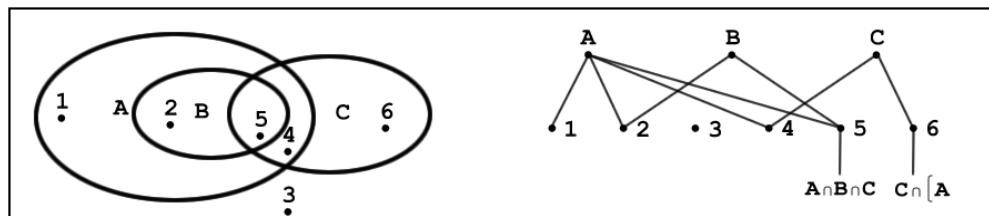


FIG. 2.13 – Représentation en machine des diagrammes de Venn

Pour démontrer le théorème T1 où A désigne l'adhérence de A (plus petit ensemble fermé contenant A) et A^* la frontière de A ($A^* = \bar{A} \cap (A)$).

Théorème T1 : $A^* = (\bar{A} - A) \cup (A \cap A)$

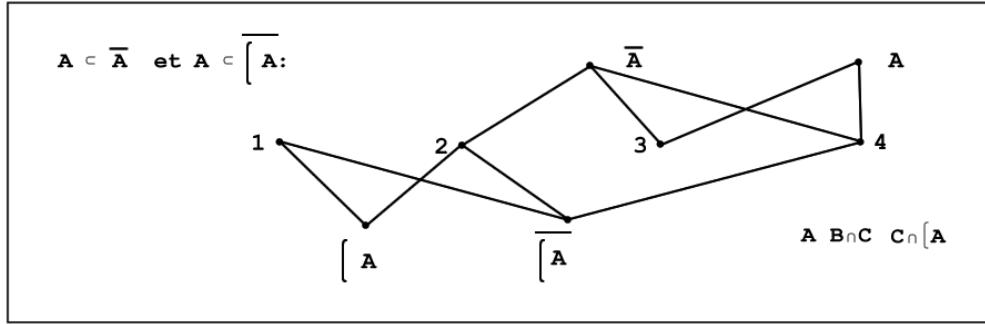
Le programme n'utilise que le graphe, le raisonnement est purement ensembliste. Tous les ensembles qui interviennent sont tout d'abord construits en respectant les inclusions dues aux définitions, en particulier :

Puis le programme construit A^* :

$$A^* = \bar{A} \cap (A) \text{ soit, sur le dessin, les atomes 2 et 4.}$$

De même $\bar{A} - A = \{2\}$, et $(A \cap A) = \{4\}$, d'où le théorème.

Bien sûr, le programme n'est pas géné, alors que le mathématicien peut le devenir, lorsque le nombre d'ensembles augmente : toutes les relations ensemblistes sont conser-

FIG. 2.14 – Construction de (A) et $\overline{(A)}$

vées dans le graphe et les démonstrations sont réduites d'autant. Plus de cent théorèmes de théorie naïve des ensembles et de topologie ont ainsi été démontrés par ce programme en des temps de l'ordre de la seconde sur IBM 43 31.

Exemple en géométrie – (Buthion 79) : « On donne un cercle (C) et deux points A et B hors du cercle. Construire une droite (D) passant par A et dont les intersections C et D avec le cercle sont équidistantes du point B ». Il nous est pratiquement impossible de raisonner sans figure. Nous supposons dans un tel cas le problème résolu et construisons la figure 2.15.

Le programme va faire de même. M nomme et construit les éléments indispensables : droite (D), milieu I de CD . Pour le programme, la figure est, en fait, un tableau où sont présents les noms des éléments avec leur nature, leur « degré de liberté », leurs « représentants » éventuels. Le degré de liberté d'un élément est un scalaire qui indique si l'élément est complètement connu, partiellement connu (ici la droite (D) qui doit passer par A), ou bien inconnu. Les représentants d'une droite sont tous les points appartenant à cette droite.

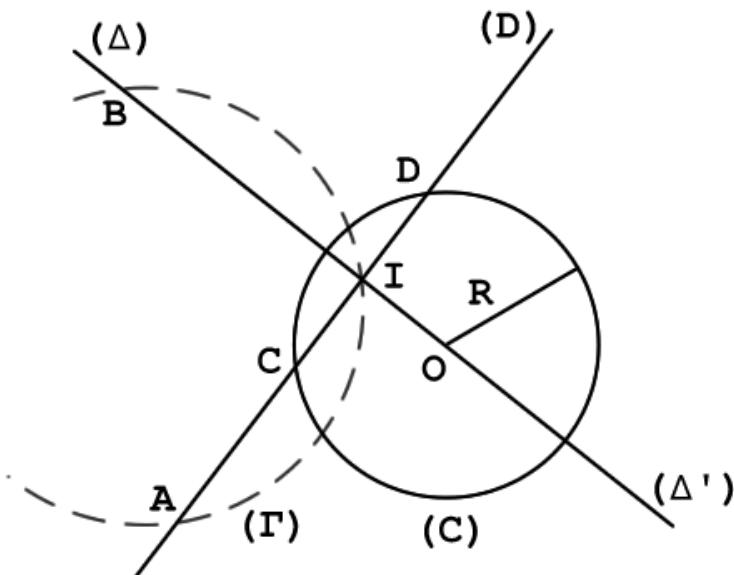


FIG. 2.15 – Un problème de construction géométrique

Dans le problème précédent, au début de la résolution, une ligne du tableau sera par exemple :

(Δ)	droite	1	B, I
(nom)	(nature)	(degré liberté)	(représentants)

Le problème est maintenant de déterminer (D) complètement. Le programme considère ici les triangles isocèles par construction COD et BCD.

OI est donc médiatrice du segment CD et BI est médiatrice du segment CO : les droites (Δ) et Δ' sont confondues, ce qui se traduit pour le programme par :

(Δ)	droite	0	(Δ') , B, I, O
(nom)	(nature)	(degré liberté)	(représentants)

Ce qui montre en particulier que I appartient à la droite complètement connue BO. En outre, l'angle AIO est droit, I appartient donc aussi au cercle, complètement déterminé, qui admet OA pour diamètre. Il est donc lui-même déterminé et la droite (D) qui passe par A et I, l'est également.

Exemple sur les propriétés des fonctions - (D. Pastre 78) : Environ cent cinquante théorèmes en théorie naïve des ensembles sur les images directes et inverses, les fonctions, les congruences, les relations d'ordre, les ordinaux ont été démontrés par le programme DATE qui travaille un peu comme le mathématicien. En particulier pour prouver le théorème T2 :

Soient trois applications $f : A \rightarrow B$, $g : B \rightarrow C$, $h : C \rightarrow A$. Si deux quelconques des trois fonctions $k_1 = h \circ g \circ f$, $k_2 = f \circ h \circ g$, $k_3 = g \circ f \circ h$ sont des surjections et la troisième une injection, alors f , g et h sont des bijections.

Le programme doit notamment montrer que si k_1 est une injection, si k_2 et k_3 sont des surjections, alors h est surjective ; pour faire cette démonstration (une des plus difficiles parmi les six démonstrations à établir pour prouver T2), le programme s'aide du dessin de la figure 2.16.

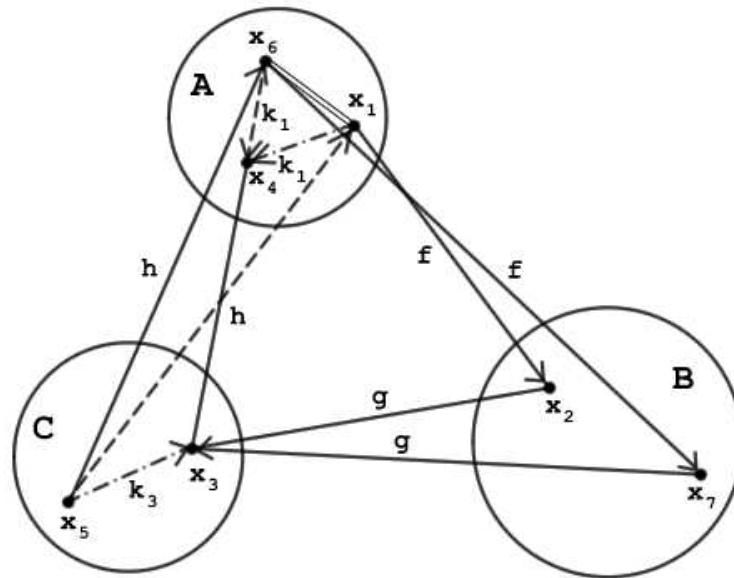


FIG. 2.16 – Un des graphes utilisés par DATE pour démontrer le théorème T2

Partant de l'élément x_1 quelconque dans A, il faut trouver x dans C tel que : $h(x) = x_1$. Le programme construit sur le dessin les points x_2 , x_3 , x_4 tels que :

$$x_2 = f(x_1), \quad x_3 = g(x_2), \quad x_4 = h(x_3)$$

Or il sait que $k_3 = g \circ f \circ h$ est une surjection, donc qu'il existe un élément x_5 de C tel que :

$$k_3(x_5) = x_3$$

Alors, DATE construit les points x_6 et x_7 liés à ce point x_5 par :

$$x_6 = h(x_5) \text{ et } x_7 = f(x_6)$$

Par construction, il sait à ce moment d'une part que :

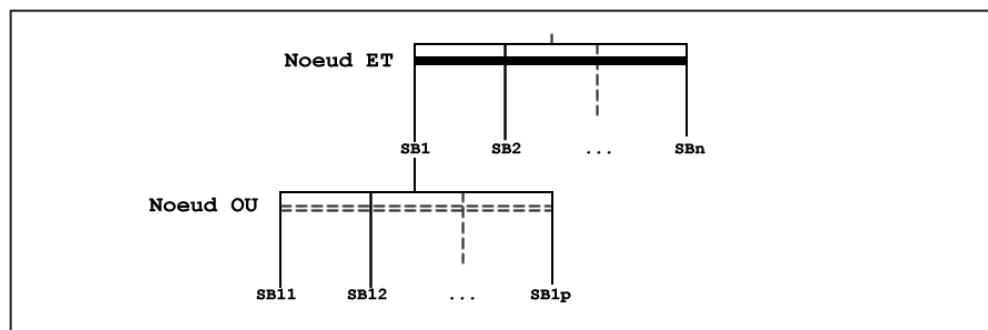
$$g(x_7) = x_3 \text{ et que } k_1(x_6) = k_1(x_7) = x_4 \text{ d'autre part ;}$$

mais k_1 est injective par hypothèse, donc les deux points x_1 et x_6 sont nécessairement confondus. Finalement on a bien construit un point x_5 qui satisfait la relation $h(x_5) = x_1$. c. q. f.d.

Ces trois exemples montrent les différents rôles que joue la figure pour aider le raisonnement et en particulier comment elle permet d'introduire de façon naturelle, mais contrôlée, de nouveaux objets.

2.10.2 Représentations graphiques en résolution automatique de problèmes :

Une méthode très générale pour résoudre un problème est la décomposition du but cherché en un ensemble de sous-but dont la conjonction satisfait les conditions demandées. Le problème est ainsi résolu lorsque chacun des sous-but pris séparément, est atteint. Mais, pour résoudre un sous-but particulier, plusieurs méthodes peuvent éventuellement être utilisées : chacune d'entre elles crée un nouveau sous-but et il suffit ici d'atteindre l'un quelconque de ces nouveaux sous-but pour atteindre le sous-but qui leur a donné naissance. La recherche est typiquement de type arborescent, on parle d'arbre « ET-OU » pour le schéma :



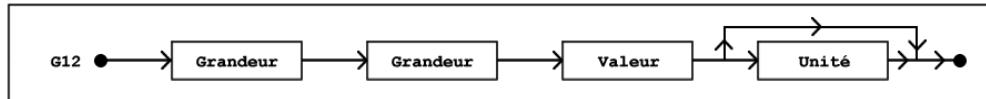
Cet arbre permet de visualiser l'effort de recherche effectif du système – homme ou machine – qui résout un problème donné. Le cas limite – idéal mais rarement rencontré – est celui où l'arbre est réduit à une seule branche ; la recherche est alors linéaire, sans essai inutile, on dit qu'on a un algorithme.

Représentation des objets : En recherche opérationnelle par exemple, les objets traités sont par nature des graphes dans maints problèmes réels. Ainsi, la recherche de la procédure optimale d'acheminement du courrier en France est tributaire du réseau

postal : les graphes des liaisons, commune par commune, via le chemin de fer, la route ou l'avion, sont des données du problème. Ainsi encore, pour minimiser le temps total d'exécution d'un ensemble de tâches soumises à des contraintes quant à leurs positions relatives dans le temps, il est commode d'utiliser un graphe représentatif (méthode PERT par exemple) qui résume ces contraintes. Un schéma comme :



exprime alors que la tâche t_1 de durée d_1 , doit être achevée pour que la tâche t_z puisse démarrer. Le schéma :



signifie que les tâches t_1 et t_z sont mutuellement exclusives et ne peuvent être exécutées simultanément.

L'intérêt essentiel de toute modélisation mathématique apparaît ici : un même modèle peut traduire des problèmes divers et seul un petit nombre de problèmes fondamentaux reste à résoudre. Ainsi un problème où seules figurent des contraintes de type (S1) est équivalent à la recherche du plus long chemin dans le graphe des tâches. Un problème où figurent seulement des contraintes de types (S2) est un problème de coloration.

2.10.3 Représentations graphiques en compréhension du langage naturel :

Les premières études de compréhension du langage naturel étaient essentiellement fondées sur une analyse syntaxique des phrases et une étude mot à mot. Elles avaient pour but principal la traduction et la documentation automatique. Les chercheurs sont aujourd'hui convaincus qu'une telle approche ne suffit pas et qu'il faut, pour comprendre un texte, donner aux programmes des informations sémantiques telles que celles qu'on trouve dans les dictionnaires et même, en outre, une masse considérable d'informations d'ordre pragmatique, qui décrivent l'univers dans lequel nous vivons, nos comportements et nos habitudes.

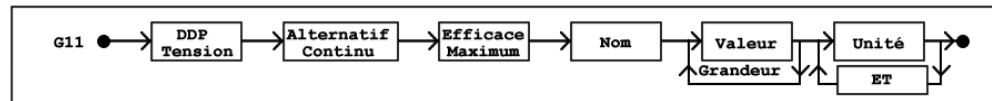
Les premières grammaires d'analyse syntaxique « context-free » de Noam Chomsky étaient facilement représentées par des règles de réécriture. Bientôt est apparue la nécessité, pour résoudre convenablement les ambiguïtés relatives par exemple aux pronoms, de suspendre momentanément le processus d'analyse de la phrase à un niveau donné pour passer à un niveau inférieur, étudier une proposition imbriquée, et revenir finalement à l'analyse interrompue.

Des graphes dit de transition ou encore ATN (« Augmented Transition Networks » voir Woods) sont habituellement utilisés actuellement pour l'analyse à la fois syntaxique et sémantique de la phrase. Les sommets de ces graphes représentent soit des mots d'une même famille sémantique, soit des graphes, d'où le caractère foncièrement récursif

d'une telle représentation. Ainsi, le programme de B. Chichetchi (Thèse 3ème Cycle. Paris 1979) comprend des problèmes d'électricité en utilisant des graphes de transition, lus en données, du type :



Ces réseaux sont des données pour le programme. Un exemple de phrase reconnue par ce graphe sera ainsi : *La fréquence de la tension V est égale à 50 Hertz*. On voit que certains mots sont sautés, que d'autres peuvent éventuellement être omis dans le texte (Hertz par exemple) sans changer l'analyse. Toute phrase correspondant à un chemin dans le réseau est a priori acceptée. G11 est le graphe des différentes descriptions possibles des caractéristiques d'un courant électrique.



Un tel graphe reconnaît ainsi la proposition : « la tension V vaut 10 volts » mais aussi : « La différence de potentiel alternative efficace est de 80 volts et l'intensité de courant 5 ampères ». A mesure qu'un texte est reconnu par de tels systèmes, il est traduit dans une représentation interne qui résume toutes les informations particulières au texte traité. Cette représentation interne est souvent appelée « réseau sémantique » (voir Simmons, Cordier, Lopez). Le réseau, vide au départ, est continuellement augmenté, modifié, corrigé, au cours d'analyse. Il peut contenir des informations certaines, comme des informations simplement probables ou encore conditionnelles.

Représentation en machine d'un transformateur dans un programme d'aide à la conception par ordinateur (M. Lopez 79) : Les sommets sont regroupés en « espaces » (encadrés), qui réunissent des unités conceptuelles ou sous-ensembles cohérents d'informations :

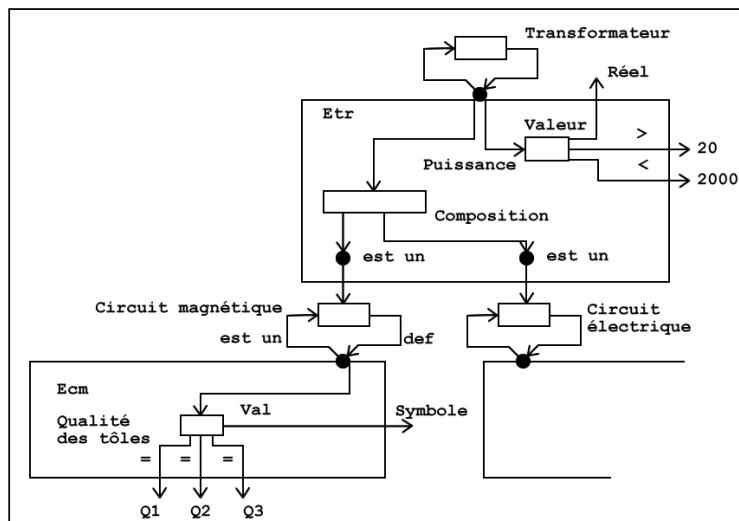


FIG. 2.17 – Réseau sémantique « partitionné »

2.11 Savoir changer de représentations

La recherche d'une bonne représentation lors de la résolution d'un problème est presque toujours une étape indispensable vers la solution. Il existe maints problèmes où nous nous servons avant tout de notre oeil et de notre main en agissant sur la seule représentation « immédiate » de l'univers tel que nous le percevons à tout instant. Aucun modèle abstrait du problème n'a alors besoin d'être posé, mais, en contrepartie, cette démarche conduit à des tâtonnements inutiles et nuit à la généralisation de l'idée qui a mené à la solution pour résoudre d'autres problèmes.

Le problème des quatre cavaliers L'échiquier et la marche des pièces aux échecs a fasciné de nombreux auteurs et est à l'origine de nombreux casse-tête. Il s'agit ici d'échanger en un nombre minimum de coups les deux cavaliers blancs et les deux cavaliers noirs à partir de la position initiale donnée par la figure 2.18 sur un échiquier 3×3 (le cavalier se déplace en un coup d'un pas horizontalement et de deux pas verticalement ou inversement).

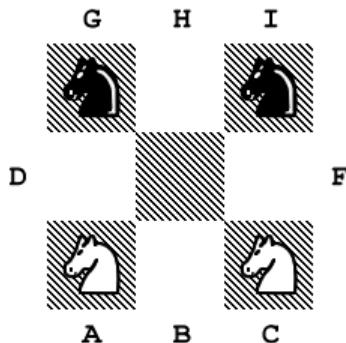


FIG. 2.18 – Les quatre cavaliers

Le premier réflexe consiste à faire quelques essais en manipulant les cavaliers sur un véritable échiquier, mais intuitivement cette représentation semble trop riche, trop détaillée ; elle utilise mal les différentes symétries du problème. Cherchant à modéliser l'énoncé dans une représentation plus générale, la représentation du plan en coordonnées cartésiennes vient à l'esprit. Elle présente ici le défaut d'alourdir encore la notation quand il s'agit de prendre en compte les mouvements, des quantités égales à plus ou moins 1 et plus ou moins 2 doivent intervenir. Mais à la réflexion, ce sont bien précisément les mouvements qui sont importants ; eux, et eux seuls, doivent être représentés commodément. L'échiquier quant à lui n'a guère ici d'importance, sa représentation physique n'est qu'un support dont l'aspect peut être changé, puisque seules comptent les liaisons entre les coups qui sont effectués par l'intermédiaire des cases. (Cf. le Rubik's cube...). Donnons donc un nom aux différentes cases comme sur la figure 2.19.

Alors nous savons que le cavalier noir en A peut se déplacer en un coup sur la case H ou encore sur la case F. Or, à son tour, un cavalier en H pourra, soit revenir en A, soit aller en C ; depuis C, poursuivant sa route, il pourra se rendre en D, puis en I, en B, en G, et finalement en F, point à partir duquel il lui sera possible de revenir en A. Dessinons alors le trajet effectué, en disposant dans l'ordre toutes les cases traversées sur une circonférence comme sur la figure 2.20 ; on y voit mieux, c'est plus agréable à l'œil.

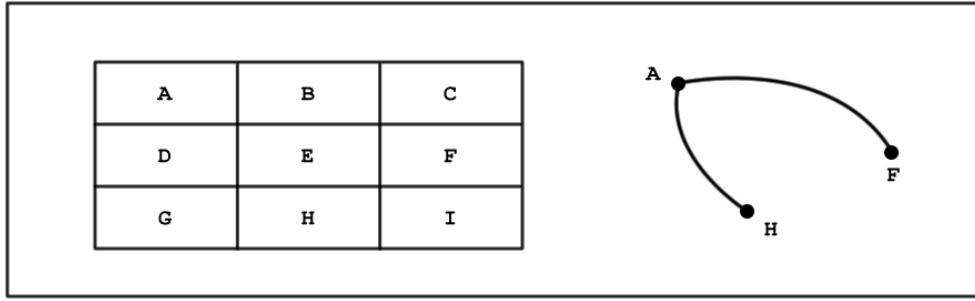


FIG. 2.19 – Noms des cases et pas du cavalier à partir de A

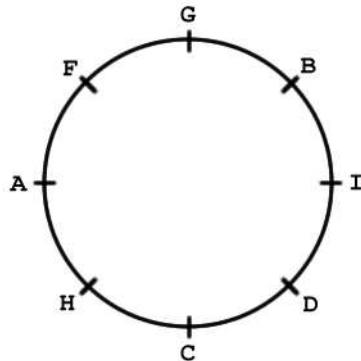


FIG. 2.20 – Le parcours du cavalier A

Sur ce dessin la case E n'apparaît pas, ce qui est bien normal puisque, en quelque sorte, elle ne faisait pas partie du problème, aucune des autres cases ne permettant de l'atteindre. Toutes les autres cases sont représentées, ce qui permet d'utiliser ce même dessin pour tous les autres cavaliers.

La nouvelle formulation du problème est maintenant d'échanger sur cette circonference des cavaliers noirs placés en A et C avec des cavaliers blancs en G et I, étant entendu que cette fois, en un coup, un cavalier se rend sur l'une des deux cases voisines sur la circonference. La solution est alors immédiate : Il suffit de faire tourner l'ensemble d'un demi-tour de façon à amener C sur G, A sur I, G sur C et I sur A. Cette rotation, qui correspond donc à quatre coups par cavalier, soit seize coups au total, est de plus minimale.

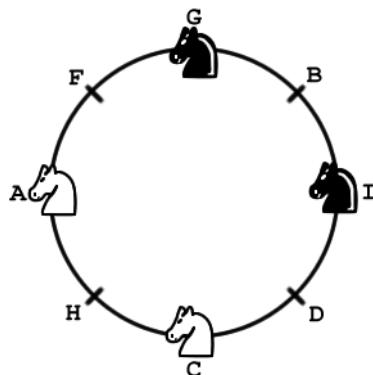


FIG. 2.21 – Solution du problème des 4 cavaliers

2.12 Le langage LISP

- a) Eléments de base.
- b) Représentation et évaluation en machine.
- c) Autres fonctions. Itération.
- d) Lambda-définition, propriétés liés aux objets.
- e) Constructions nouvelles, F expressions. Macros.

L'importance de la manipulation symbolique est telle, en Intelligence artificielle, que dès 1956 l'équipe Newell, Shaw et Simon invente IPL (Information Processing Langage) qui permet de manipuler des listes. IPL est un des ancêtres de LISP (List programming) qui fut forgé par John McCarthy au MIT en 1960. Dans la famille de tous les langages de programmation jusqu'à ce jour, LISP présente un aspect unique et un goût spécial. Certains programmeurs en I.A. ne jurent que par lui notamment aux Etats Unis où la norme est aujourd'hui COMMONLISP (Steele, 1984). Il est assurément dommage que son usage ne soit pas plus répandu dans le monde général de l'informatique tant il est **rigoureux, concis et agréable**.

LISP fournit donc automatiquement l'outillage de la représentation arborescente dont nous avions parlé auparavant. Comme celle-ci est au cœur de la manipulation symbolique, qui est elle-même la matière première en Intelligence artificielle, il n'est guère étonnant que **bien des programmes d'I.A. soient écrits en LISP**.

L'outillage de listes, une fois assimilé, se programme cependant très facilement dans les langages évolués : FORTRAN, PLI, PASCAL, ADA, etc.

2.12.1 Eléments de base du langage LISP

Syntaxe *S-expression* : les expressions LISP sont aussi appelées des S-expressions ou S-exp. Une S-expression est soit un **atome**, soit une **liste**.

Un atome est un nom alphanumérique quelconque (éventuellement un nombre). Une liste est un *intérieur* entouré de parenthèses. La syntaxe sous forme de Naur-Backus est la suivante :

```
<S-Exp>  :=  <atome>
          :=  <list>
<liste>  :=  (<intérieur>)
<intérieur>  :=  <vide>
              :=  <S-exp>
              :=  <S-exp><intérieur>
<vide>  :=
<atome>  :=  chaîne alphanumérique sans blanc,
              ni caractères spéciaux : (, ), :, .. , '
```

Toute S-expression est ainsi un assemblage bien parenthésé d'atomes ; exemple :

((ROI DAME) ((TOUR) (CAVALIER FOU)) PION)

Fonctions élémentaires Comme dans tous les langages de programmation, certains atomes prédéfinis sont des fonctions dont les arguments sont les S-expressions qui suivent. Mais un argument peut lui-même être une fonction qu'il faut aussi évaluer. L'évaluateur doit lors pouvoir faire la différence entre la valeur L et la S-expression à laquelle on a donné le nom symbolique L. Pour cela, on fait précédé, dans le premier cas, l'expression d'une apostrophe et on écrit : 'L, ou sous forme développée (QUOTE L). L'apostrophe inhibe l'évaluation et la S-expression L est prise telle quelle.

Pour donner un nom à une expression on utilise la fonction SETQ :

$$(\text{SETQ } <\text{atome}> <\text{S-exp}>) \rightarrow <\text{S-exp}> \text{ non évaluée.}$$

où la flèche signifie *renvoie pour valeur*. D'une façon générale, quand on entre en machine une S-exp, celle-ci est immédiatement évaluée :

$$(\text{SETQ } \mathbf{G} \ 'F) \rightarrow \mathbf{F}.$$

En outre SETQ attache ce résultat à $<\text{atome}>^\dagger$: $\mathbf{G} \rightarrow \mathbf{F}$

Deux fonctions fondamentales CAR et CDR permettent de manipuler les listes non vides.

(CAR '<liste non vide>) → première sous-expression de liste, parenthèses ôtées :

$$\begin{aligned} (\text{CAR } '(A B C)) &\rightarrow A \\ (\text{CAR } '((A)B)) &\rightarrow (A) \\ (\text{CAR } ' A) &\rightarrow \text{erreur : } A \text{ n'est pas une liste} \\ (\text{CAR } '()) &\rightarrow \text{erreur ou NIL selon les dialectes LSIP} \end{aligned}$$

(CDR '<liste non vide>) → liste composée du complément entre parenthèses du CAR de la liste argument :

$$\begin{aligned} (\text{CDR } '(A B C)) &\rightarrow (B C) \\ (\text{CDR } '((A) B)) &\rightarrow (B) \\ (\text{CDR } 'A) &\rightarrow () \quad ; \text{ par convention} \\ (\text{CAR } (\text{CDR } '(A B C))) \\ &\xrightarrow{\hspace{1cm}} (B C) \\ &\xrightarrow{\hspace{1cm}} B \end{aligned}$$

Les notations étranges CAR et CDR ont une origine historique : ils désignaient, au moment de la première implémentation de LISP sur IBM 7094, les contenus respectifs de l'adresse register et du décrément register.

Les deux fonctions et leurs différentes compositions sont fort utiles et donnent lieu à des abréviations classiques :

$$\begin{aligned} (\text{GADR } S) &\equiv (\text{CAR } (\text{CDR } S)) \\ (\text{CADR } '(A B C)) &\rightarrow B \\ (\text{CADDAR } S) &\equiv (\text{CAR } (\text{CDR } (\text{CDR } (\text{CAR } S)))) \end{aligned}$$

et ainsi de suite.

La fonction CONS reconstruit ce que CAR et CDR ont séparé :

$$\begin{aligned} (\text{CONS } <\text{S-exp}> <\text{liste}>) &\rightarrow \text{liste} \\ (\text{CONS } 'A '(B C)) &\rightarrow (A B C) \end{aligned}$$

Le second argument de CONS peut être la liste vide (), également notée NIL.

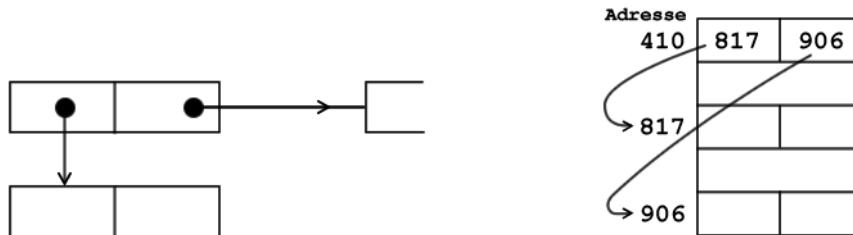
$(\text{CONS } 'A \text{ NIL}) \rightarrow (A)$

Quelle que soit la S-exp S, on a : $(\text{CONS } (\text{CAR } S) (\text{CDR } S)) = S$

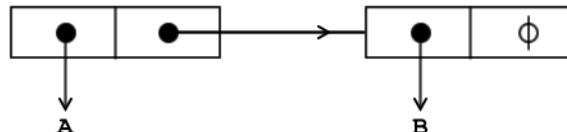
Donnons maintenant la représentation habituelle en machine des objets LISP et l'algorithme d'évaluation des S-exp.

2.12.2 Représentation et évaluation

Pour un interpréteur LISP les cases mémoires sont couplées deux à deux et chacune d'elles contient un pointeur que nous convenons de représenter par une flèche. Le schéma de gauche correspond à la situation physique de droite :



Les deux pointeurs jouent respectivement le rôle de premier fils et de premier frère. L'instruction (SETQ L1 '(A B)) crée ainsi la structure :



Avec les conventions suivantes :

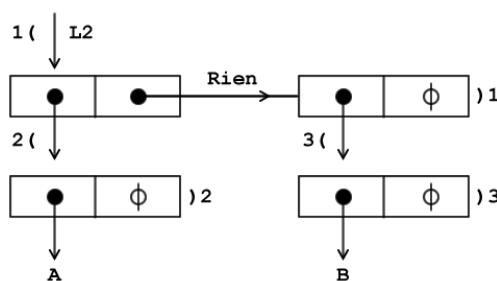
L'absence de pointeur est indiquée par le symbole ϕ .

Les **atomes** sont traités séparément (marqueur spécial).

Le **CAR** d'une S-exp est ce sur quoi pointe le premier fils.

Le **CDR** d'une S-expr est ce sur quoi pointe le premier frère mis entre parenthèses.

Pour retrouver une S-exp à partir de son schéma il faut parcourir celui-ci en suivant les pointeurs gauches au plus profond et sinon les pointeurs droits (parcours *antérieur*). On écrit, pour chaque flèche descendante qui n'arrive pas sur un atome, une parenthèse ouvrante, pour chaque symbole ϕ une parenthèse fermante, pour tout atome l'atome lui-même. Ainsi au schéma :



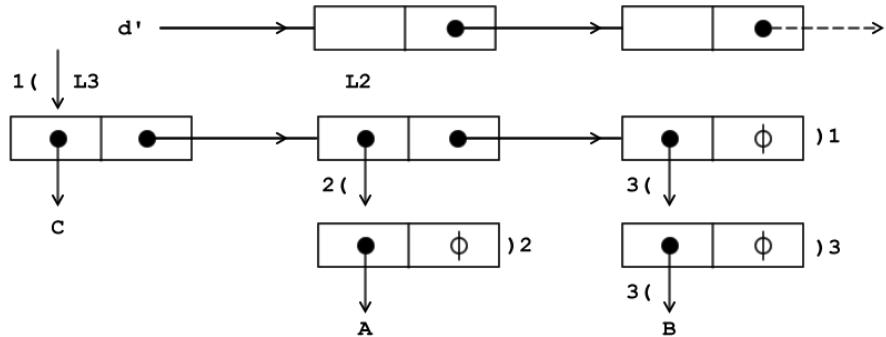
correspond la liste L2 : ((A) (B)).

De plus, à tout moment, l'interpréteur gère la liste des cellules disponibles dont la tête est d :



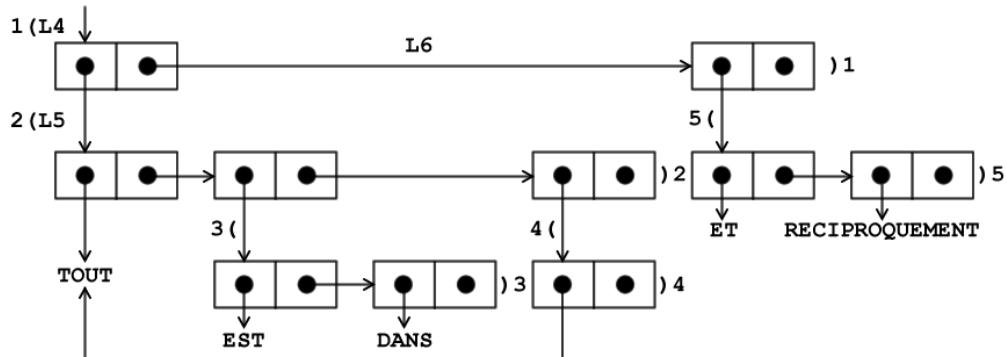
A une forme comme : (SETQ L3 (CONS 'C L2))
correspond alors :

si l'atome C n'existe pas, le déplacement du pointeur d d'un cran et la fabrication de C, la mise en place du pointeur frère de C vers la tête de la liste L2 ; soit le schéma :



On a de même pour la liste L4 la représentation suivante :

$$L4 = \underbrace{((TOUT)}_{L7} \underbrace{(EST\ DANS)}_{L8} \underbrace{(TOUT)}_{L9} \underbrace{(ET\ RECIPROQUEMENT))}_{L6}$$



$$(\quad (TOUT \quad (EST \quad DANS) \quad (TOUT \quad) \quad) \quad (ET \quad RECIPROQUEMENT) \quad) \\ 1 \quad 2 \qquad \qquad 3 \qquad \qquad 3 \quad 4 \qquad \qquad 4 \quad 2 \quad 5 \qquad \qquad \qquad 5 \quad 1$$

Evaluation d'une liste La figure 2.22 donne le principe de l'évaluation en LISP : la fonction EVAL. Le système travaille, bien sûr, sur la représentation interne : les parenthèses ne figurent plus. Une **exécution** est une **évaluation** de toute S-expression entrée. Celle-ci est parcourue en suivant les pointeurs fils tant qu'il y en a, en prenant ensuite le dernier frère rencontré, puis les fils de celui-ci et ainsi de suite, jusqu'à ce qu'une valeur terminale puisse être calculée.

L'évaluation est alors répercutée en remontant les pointeurs vers l'expression d'origine. Entre temps, les morceaux de S-exp qui attendent la valeur de leurs arguments sont appelés des « *quasars* ».

Procédure EVAL(S) avec S = S-expression LISP en représentation interne

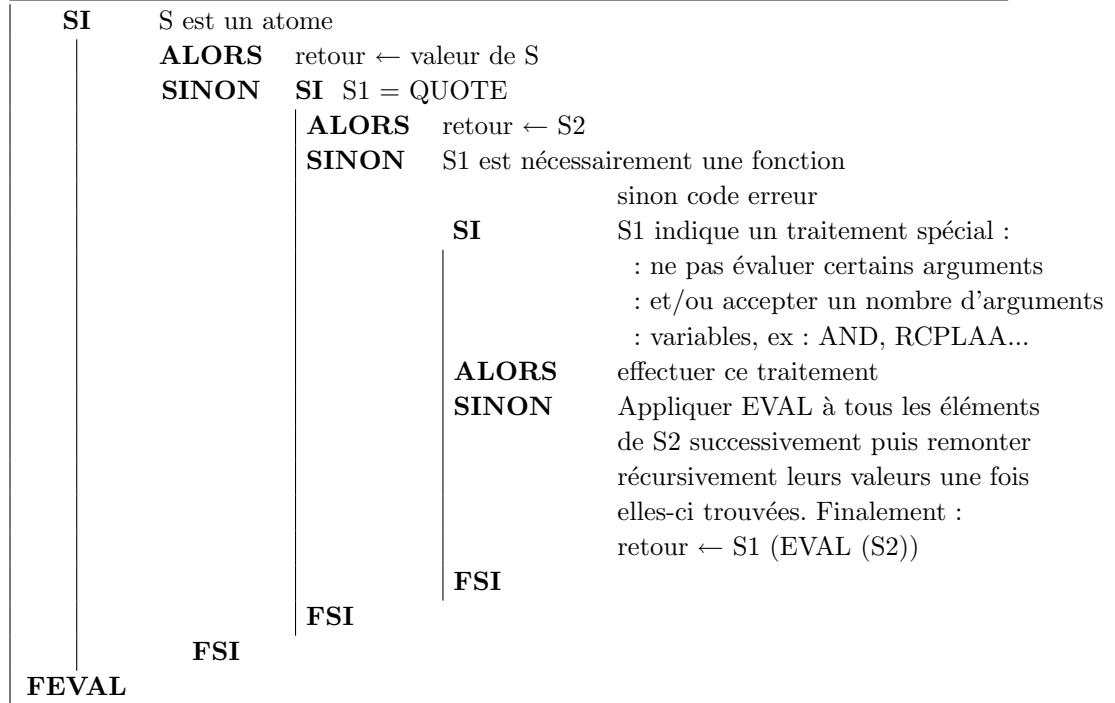


FIG. 2.22 – Principe de l'évaluation en LISP : la fonction EVAL. Ce schéma suppose que 'L a toujours été auparavant développé en (QUOTE L). S1 désigne le premier fils de la S-expression S et S2 les éléments de S ensemble des frères de S1

Remarque importante : **Un programme LISP est un S-exp.**

En particulier, en LISP, les données ont même structure que les programmes, ce sont aussi des S-expressions. Le résultat d'un programme LISP est lui-même, par définition des évaluations de toutes les fonctions de base, une S-expression donc un programme. Cette caractéristique particulière permet d'écrire des programmes qui se modifient eux-mêmes. Dans EVAL, les nombres entiers : -4 ou réels : 3.1416, la liste vide NIL ; et le symbole T (pour true) sont leurs propres valeurs : Il est inutile de les faire précédé d'une apostrophe.

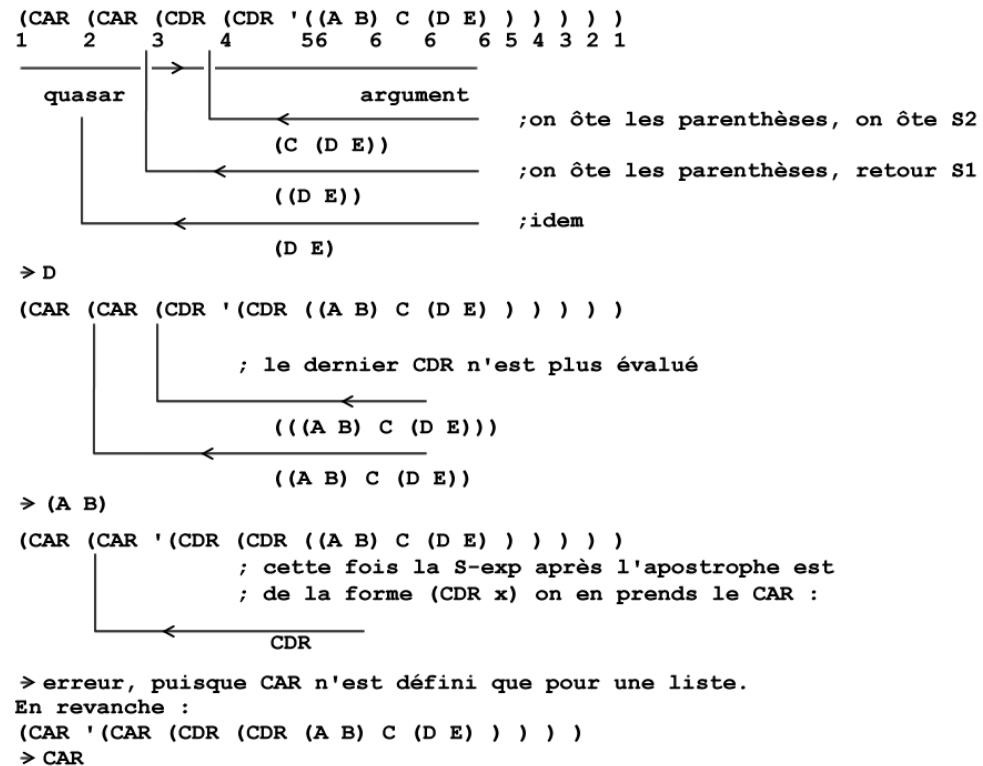
•Exemples d'évaluations :

(CAR '(A)) → A	
(CDR '(A)) → NIL	
(CAR 'A) → erreur	
(CDR 'A) → NIL	; par convention.
(CAR '((A))) → (A)	
(CDR '(A (BC))) → ((BC))	; attention !
(CAR '(CDR (A B C))) → CDR	; l'apostrophe inhibe l'évaluation.
(CAR (CDR (A B C))) → erreur	; A devrait être une fonction.

EVAL peut être appelé sur elle-même :

(SETQ A 'B) → B
 (SETQ A 'C) → C
 A → B
 B → C
 (EVAL A) → C

; plus difficile :



2.12.3 Autres fonctions

2.12.3.1 Les prédictats (valeur T ou NIL)

(ATOM <S-exp>) → T si et seulement si la valeur de S-exp est un atome ou une fonction, sinon NIL.

(ATOM 'A) → T
 (ATOM NIL) → T
 (ATOM 'CAR) → T
 (ATOM '(A B)) → NIL

(NULL <S-exp>) → T si et seulement si la valeur de S-exp est NIL, NIL sinon.

(NULL (CDR '(ATOME))) → T

(EQUAL <S-exp1> <S-exp2>) → T si et seulement si les deux valeurs sont égales.

(EQ <S-exp1> <S-exp2>) → T si et seulement si les deux valeurs sont une seule et même liste physique (au même emplacement).

(EQUAL '(A B) '(A B)) → T ; tandis que
En particulier : (EQ '(A B) '(A B)) → NIL ; car la liste (A B) a été créée deux fois,
à coup sûr à deux places différentes.

(AND <S-exp >*) : nombre quelconque d'arguments qui sont évalués de gauche à droite si NIL est rencontré alors NIL est la valeur retournée sinon la valeur de la dernière S-exp est retournée

(OR <S-exp >*) : de même ici si autre chose que NIL est rencontré, cette valeur est retournée. Sinon NIL est retourné.

(NOT <S-exp>) → T si et seulement si la valeur de la S-exp est NIL.

2.12.3.2 Les fonctions

Le choix entre plusieurs possibilités se fait grâce à la fonction COND dont la syntaxe est :

(COND (<t₁> <r₁>)
<t₂> <r₂>
...
<t_n><r_n>))

Les *t_i* (*t* pour test) et les *r_i* (*r* pour résultat) sont des S-exp. Les *t_i* sont évalués dans l'ordre croissant des indices. Dès qu'un *t_i* prend une valeur différente de NIL, COND évalue *r_i* et retourne la dernière S-exp de *r_i*. Si tous les *t_i* valent NIL alors NIL est la valeur finale.

Certains *r_i* peuvent être omis. Dans ce cas si *t_i* est encore le premier *t* différent de NIL, COND renvoie la valeur de *t_i*.

- **Fonctions de construction de listes :**

Les deux fonctions APPEND et LIST reconstruisent des listes comme le faisait CONS :

(APPEND <listel><liste2>... <listen>) → la liste formée de listel, liste2,... listen auxquelles ont été retirées les parenthèses au premier niveau.

exemple : (APPEND '(A) NIL '((B)(C))) → (A (B)(C))

(LIST <S-exp1> <S-exp2>...<S-expn>) → (S-exp1 S-exp2 ... S-expn)

Cette fois les parenthèses subsistent : en outre, LIST, contrairement à APPEND, admet un atome pour argument : (LIST '(A) NIL '((B)(C))) → ((A) NIL ((B) (C)))
(LIST '(A) NIL) → (A NIL)

(LENGTH <S-exp>) → nombre de frères au premier niveau dans S-exp.

(LENGTH '((A B) C (D))) → 3

(REVERSE <S-exp>) → la S-exp formée des frères au premier niveau de la S-exp d'entrée dans l'ordre inverse.

(REVERSE '(A (B C) D)) → (D (B C) A).

(SUBST <S-exp1><S-exp2> <S-exp3>) →

le résultat de la substitution dans S-exp3 de toutes les occurrences de S-exp2 par S-exp1. La valeur de S-exp2 doit être un atome.

(SUBST 'A 'B'(A B C)) → (A A C) **MEMBER** (<atome> <list>) → NIL si et seulement si la valeur d'<atome> n'apparaît pas comme frère au premier niveau dans <liste> : sinon toute la sous-liste à partir de la première occurrence d'<atome> est retournée. **Exemple :** (SETQ M '(A B (C D) E))

(MEMBER 'F M) → NIL

(MEMBER 'C M) → NIL

(MEMBER 'B M) → (B (C D) E)

(RPLACA <Liste> <S-exp>) →

remplace le CAR de la liste par S-exp dans <liste> elle-même.

(SETQ G '(A B C)) → (A B C)

(RPLACA (CDR G) 'D) → (D C)

Et de plus RPLACA induit un effet de bord sur la liste d'entrée elle-même : G → (AD C) **RPLACD** est la fonction symétrique agissant sur le CDR du premier argument.

2.12.3.3 Les fonctions numériques :

Par cohérence avec ce qui précède elles s'écrivent toujours en **notation polonaise préfixée**. La signification des fonctions suivantes est évidente :

(PLUS 2 3)	→ 5	; PLUS peut avoir un nombre
(PLUS 1 2 3 4 5)	→ 15	; quelconque d'arguments
(DIFFERENCE 5 2)	→ 3	
(TIMES 4 5)	→ 20	
(QUOTIENT 8 2)	→ 4	
(MAX 5 8 7 6)	→ 8	
(MIN 5 8 7 6)	→ 5	
(ADD1 6)	→ 7	
(SUB1 7)	→ 6	
(SQRT 16)	→ 4	
(EXPT 2 3)	→ 8	
(MINUS 3)	→ -3	
(ABS 3)	→ 3	
((MAX (MIN 2 4 6)) (MAX 3 5 2))	→ 5	
(ATOM (PLUS 1 2 3))	→ T	

car la S-exp argument est évaluée à 6 qui est bien un atome.

Trois nouveaux prédictats ont pour arguments des nombres :

(LESSP N1 N2)	→ T	ssi valeur N1 < valeur N2
(GREATERP N1 N2)	→ T	ssi valeur N1 > valeur N2
(ZEROP N1)	→ T	ssi valeur N1 = 0.

En outre :

(REMAINDER N1 N2) → reste de la division de la valeur de N1 par la valeur de N2.

Dans les lignes précédentes si les valeurs de N1 et N2 ne sont pas des nombres le système donne un message d'erreur. Il est possible de tester si une S-exp est un nombre ou non au moyen du prédictat.

(NUMBERP <S-exp>) → T ssi la valeur de la S-exp est un nombre.

2.12.3.4 L'itération : MAPCAR et APPLY

(MAPCAR '<fonction> <S-exp>') → la S-exp formée de l'application du premier argument (qui doit être une fonction connue du système) à tous les éléments de <S-exp>.

(MAPCAR 'ADD1 '(4 8 3)) → (5 9 4)

Le nombre d'arguments de <fonction> doit être égal à 1 pour pouvoir être mis en correspondance avec les frères au premier niveau de <S-exp>.

(APPLY '<fonction> <S-exp>') → cette fois, la valeur de <fonction> appliquée à la valeur de <S-exp> :

(SETQ A '(4 5 8 D)) → (4 5 8 1) (APPLY 'PLUS A) → 18 Cette fonction est nécessaire car l'essai (PLUS A) donne une erreur, puisque PLUS s'attend à recevoir des arguments qui sont des nombres.

Ces deux fonctions permettent également d'itérer une action sur tous les éléments d'une liste.

2.12.4 Lambda définition et propriétés

2.12.4.1 Lambda définition

La lambda notation due à Church (1930) permet d'utiliser une fonction localement sans même lui donner de nom :

((LAMBDA (X)) <S-exp1>)<S-exp2> → la valeur de <S-exp1> dans laquelle toutes les occurrences de X ont été remplacées par la valeur de <S-exp2>.

X est la *variable*

S-exp1 est le *corps* de la lambda fonction.

S-exp2 est *l'argument*.

((LAMBDA (X) (EQUAL X 'CRAYON)) 'CRAYON) → T

(MAPCAR '(%(LAMBDA ())) (*|||)) '(1 2 3 4) → (1 8 27 64)

Notons, dans ce deuxième exemple, l'obligation de mettre une apostrophe devant la lambda fonction.

2.12.4.2 Propriétés attachées aux atomes : PUTPROP et GET

Les tableaux, les indices n'ont pas de sens en LISP. Pour parler d'une propriété liée à

un objet et de sa valeur on utilise la fonction PUTPROP : (**PUTPROP**<atome1><liste> <atome2>).

Elle a pour effet « d'attacher » la propriété dont le nom est <atome2> à l'objet <atome1> en lui attribuant pour valeur celle de <liste>. Ainsi :

(PUTPROP 'PIERRE '(JEAN JACQUES) 'ENFANTS) → (JEAN JACQUES) et attribue physiquement (par pointeurs) à « Pierre » les « enfants » Jean et Jacques. Pour savoir maintenant si un objet possède une propriété donnée, on a recours à GET (GET<atome1> <atome2>) → valeur de la propriété <atome2> au point <atome1> si elle existe, NIL sinon. (GET 'PIERRE 'ENFANTS) → (JEAN JACQUES)

2.12.5 Constructions nouvelles

LISP permet de définir de nouvelles fonctions grâce à « DEFUN » dont la syntaxe est :

(DEFUN <atome> (<p₁> <p₂>... <p_n>) <S-exp>) où <atome> est le nom de la fonction qu'on désire définir, les *p_i* sont les paramètres formels, ou variables liées, et <S-exp> est le corps de la fonction.

Exemple : Reconstruisons d'abord deux fonctions simples comme si elles n'existaient pas :

```
(DEFUN LONGUEUR (L))
  (COND ((NULL L) 0) ; si L est NIL alors 0
        (T (ADD1 (LONGUEUR (CDR L))))
          ; Sinon on ajoute 1 en ôtant le CAR.
          ; on a reconstitué LENGTH
→ LONGUEUR ; le système restitue le nom de la fonction.
Le cas de REVERSE qui renverse l'ordre des frères au premier niveau n'est pas plus compliqué.
```

```
(DEFUN REVERSE (S)
  (COND ((NULL S) S) ; si L est NIL alors S.
        (T (APPEND (REVERSE (CDR(S)) HLIST(CAR S)))
→ REVERSE
```

- Fabriquons l'union de deux ensembles représentés en LISP par des listes formées exclusivement d'atomes :

```
(DEFUN UNION ( E F) ; le résultat sera dans F
  (COND ((NULL E) F))
        ((APP (CAR E) F) (UNION (CDR E) F))
          ;si le premier élément e de E appartient à F, on progresse
          ;dans l'ensemble E puisque e est déjà dans F.
          ;APP reste une fonction à définir.
          ;sinon il faut ajouter à F et progresser de la même façon.
  (T (UNION (COR E) (CONS (CAR E) F)))) → UNION
```

Fabriquons la fonction à deux arguments APP :

```

(DEFUN APP (A X) ; l'atome A est-il dans X ?
  (COND ((NULLX) NIL) ; non si l'ensemble X est vide
        ((EQUAL A (CAR X)) T) ; oui ici
        ; sinon on progresse dans l'ensemble X :
        (T (APP A (CDR X)))))
→ APP
(UNION '(A B C D E) '(Z U B D Y)) → (E C A Z U B D Y)

```

- Soit la suite numérique définie par :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{sinon} \end{cases} \quad (2.5)$$

Il vient en LISP pour calculer la liste des u_i :

```

(DEFUN PIMP (u)
  (PRINT u) ; on imprime les valeurs successives
  (COND ((EQUAL u 1) NIL) ; arrêt si  $u = 1$ 
        (ZEROP (REMAINDER u 2)) ; cas  $u$  pair :
        (PIMP (QUOTIENT u 2)))
        (T (PIMP (ADD1 (TIMES u 3))))))
→ PIMP

```

$(\text{PIMP } 17) \rightarrow 17 \ 52 \ 26 \ 13 \ 40 \ 20 \ 10 \ 5 \ 16 \ B \ 4 \ 2 \ 1$ • Ecrivons une fonction FILTRE (*« pattern matching »* en anglais), qui compare deux chaînes de caractères F (comme forme) et E (comme expression). Cette fonction est un cas particulier de l'algorithme *« d'unification »* que nous renconterons au chapitre III.

Les caractères de F et E sont comparés un à un et de gauche à droite. S'ils sont identiques on continue. Mais, en outre, certains caractères de F peuvent être des caractères spéciaux : + et *. Et l'on veut que, quand on rencontre + dans F, alors tout caractère soit accepté dans E ; exemple $(\text{FILTRE } '(A + C) '(A B C)) \rightarrow T$

Quand on rencontre * dans F, alors un nombre quelconque de caractères est accepté dans E : exemple $(\text{FILTRE } '(* C) '(A B C)) \rightarrow T$

Il vient donc :

```

(DEFUN FILTRE (F E)
  COND ((AND ((NULL F) (NULL E)) T) ; F et E vides
        ((OR ((NULL F) (NULL E)) NIL) ; une seule vide
         ((OR ((EQUAL(CAR F) (CAR E))
                ((EQUAL (CAR F) '+)) ; cas du plus
                ((FILTRE (CDR F) (CDR E))))
                ((EQUAL (CAR F) '*)) ; cas de l'étoile
                (COND ((FILTRE (CDR F) (CDR E))
                      ((FILTRE F (CDR E))))))
                (T NIL))) ; sinon échec
→ FILTRE

```

Dans le dernier COND, les parties résultats sont vides : on sait qu'alors le résultat est la partie test elle-même. Dans le premier cas, l'étoile est sautée comme si c'était un + ; dans le second, on avance dans E sans avancer dans F et on réappelle FILTRE : l'étoile a « absorbé » le caractère.

Exemple : $(\text{FILTRE } '(*B K + M * Z)'(B A O B A B K L M Y Z)) \rightarrow T$

La forme LET permet d'affecter des valeurs à des identificateurs qui servent ensuite d'arguments pour des expressions. LET permet notamment de ne pas calculer plusieurs fois la même entité.

(LET ((nom1 S-exp1)
 (nom2 S-exp2)
 ...
 (nomn S-expn))
 <expressions>^{*}) → valeurs des expressions où les noms i
 ont été remplacés par les valeurs des S-exp.

Exemple :

```
(DEFUN PQCD NI N2)
  ((LET ((M (REMAINDER N1 N2)))
    (COND ((ZEROP M) N2)
      (T (PGCO N2 M)))))
```

2.12.5.1 Fonctions FEXPR et MACRO

Une fonction de type FEXPR n'évalue pas son argument. La syntaxe est :

```
(DEFUN <Nom fonction FEXPR (paramètre) <corps>)
```

Exemple :

```
(DEFUN IMPRIME FEXPR (P) (PRINT P))
→ IMPRIME
  (IMPRIME CETTE PHRASE)
→ (CETTE PHRASE)
→ T
```

Les fonctions QUOTE, SETQ, COND sont de type FEXPR.

Une fonction de type MACRO traduit d'abord le corps de la fonction qui est, seulement ensuite, évaluée.

```
(DEFUN <nom fonction> MACRO (par) <corps>)
```

Exemple : fabrication du SI ... ALORS ... SINON classique.

Le paramètre sera la liste (SI TEST ACTION1 ACTION2). Comme les expressions TEST, ACTION1, ACTION2 sont inconnues lors de l'appel, nous utiliserons l'instruction de substitution SUBST pour placer ces expressions convenablement dans un COND qui sera finalement évalué :

```
(DEFUN SI MACRO (X)
  (SUBST (CADR X) 'TEST
    (SUBST (CADDR X) 'ACTION1
      (SUBST (CADDR X) 'ACTION2
        '(COND (TEST ACTION1)
          (T ACTION2))))))
→ SI
```

Rappel : (SUBST x y z) substitue *x* à toutes les occurrences de *y* dans l'expression *z*. Ici chaque SUBST agit donc sur toute la suite du programme. La MACRO ci-dessus définit effectue une transformation de son programme par lui-même.

2.13 Les graphes

Une généralisation de la représentation symbolique des expressions sous forme d'arbres conduit à la notion de *graphe*. Un graphe G est un ensemble X d'objets, appelés *sommets*, sur lequel est donnée une relation binaire. On note alors cet ensemble $G = (X, U)$. La relation R est donnée par l'ensemble des couples de sommets qu'elle relie.

$$\begin{aligned} X &= \{1, 2, 3, 4, 5\} \\ X &= \{(1, 1), (1, 2), (1, 3), (1, 4), (2, 4), (3, 4), (3, 5)\} \end{aligned} \tag{2.6}$$

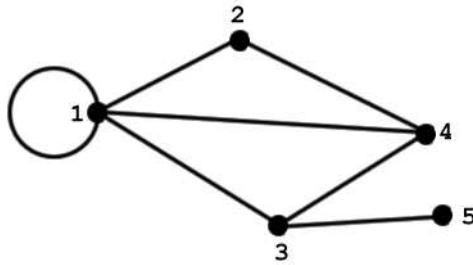


FIG. 2.23 – Exemple de graphe

Si la relation R est symétrique (ce que nous supposons dans la figure 2.23), les couples de R sont *non orientés* et sont appelés *arêtes*. Dans le cas contraire, l'ordre des sommets u et v dans la paire (u, v) de R importe et ces paires de sommets sont appelés des *arcs*. L'interprétation la plus naturelle d'un graphe est un schéma de liaisons entre points : réseaux téléphoniques ou routiers (en général non orientés) ou électriques (orientés, le sens est imposé par la source du courant).

Définitions : Un graphe $G' = (Y, V)$ est un *graphe partiel* d'un graphe $G = (X, U)$ si et seulement si $Y = X$ et $V \subset U$. C'est un *sous-graphe* si et seulement si $Y \subset X$ et si $V \subset U - W$ avec $W = \{(v, w) | v \in (X - Y) \text{ ou } w \in (X - Y)\}$: certains sommets et tous leurs arcs incidents ont été supprimés. Si $u = (x, y)$ appartient à U , x est appelé *extrémité initiale* de u ; y est l'*extrémité terminale* : x est un *prédecesseur* de y et y un *successeur* de x .

Le *degré* d'un sommet est son nombre de *voisins*, c'est-à-dire le nombre de sommets qui sont ses prédecesseurs ou ses successeurs.

Un *chemin* d'un sommet a à un sommet b dans un graphe $G = (X, U)$ est une suite finie de sommets du graphe G , (c_1, c_2, \dots, c_n) , telle que $c_1 = a$, $c_n = b$ et $\forall i (c_i, c_{i+1})$ appartient à U . Si G n'est pas orienté il suffit, dans la définition ci-dessus, que soit (c_i, c_{i-1}) , soit (c_i, c_{i+1}) appartenante à U et on parle alors de *chaîne* de a à b .

Un *circuit* (respectivement un *cycle*) est un chemin (respectivement une chaîne) qui se referme sur lui-même : $a \equiv b$.

Lorsque, pour tout couple a et b de sommets de G , il existe une chaîne de a vers b , G est dit *connexe* (*fortement connexe*, s'il existe un chemin).

Les graphes permettent de représenter de très nombreuses situations courantes : outre les divers réseaux déjà évoqués, ils servent à modéliser maintes formes de **communications**, mais aussi des relations de **parenté**, les **flux** de matériaux ou d'information,

les **formules chimiques** et les expressions symboliques en général puisque les arbres sont des graphes particuliers [les graphes de fonction en analyse ne sont autres que des cas particuliers également où l'on dessine l'ensemble des points $(x, f(x))$].

Nous verrons aux chapitre IV, V et VIII de nombreux exemples de cette utilité des graphes et de la force des modélisations qu'ils autorisent.

Chapitre 3

Les Systèmes Formels

3.1 Introduction

La logique formelle se propose d'élaborer une théorie des raisonnements valides. Elle étudie pour cela les raisonnements dans leur forme, non dans leur sens : les éléments du discours sur lesquels porte le raisonnement peuvent être arbitrairement substitués par d'autres partout où ils apparaissent. Dans le syllogisme bien connu, attribué généralement à Aristote, mais dû en réalité à Guillaume d'Occam (1349) :

« *Les hommes sont mortels*
(S) *Socrate est un homme*
 donc Socrate est mortel »

Les deux occurrences de chacun des mots : homme, mortel, Socrate, peuvent être remplacées par n'importe quel mot, le raisonnement restera formellement valide. Un moyen simple de **raisonner sur ces raisonnements** est, dès lors, de remplacer ces mots substituables par des symboles. Ainsi, dans l'exemple cité, le modèle abstrait du raisonnement est :

« *Si tout x est y*
Si z est un x
Alors z est y »

C'est dans la syllogistique, fondée par Aristote, que l'on rencontre effectivement pour la première fois, dans l'*Organon*, un usage systématique de tels symboles substituables, plus tard appelés *variables*.

Mais, dans l'essai de systématisation du raisonnement tel qu'il est exprimé notamment dans le langage usuel, l'introduction des variables ne suffit pas : les mots de liaison : si, alors, ou, et, car, donc... et les verbes : est, appartient, implique... jouent un rôle fondamental. Ils constituent les **articulations** du raisonnement formel et, en tant que tels, ne peuvent être remplacés par autre chose : le syllogisme donné (S) ne tient plus si le deuxième « si » est remplacé par un « ou bien ». Ces symboles non substituables sont appelés *opérateurs*.

Remarquons que le passage d'un énoncé en français à son formalisme logique n'est pas toujours immédiat : le « ou » signifie tantôt « ou bien » (ou *inclusif*), tantôt « ou au contraire » (ou *exclusif*) ; le verbe « être » traduit tantôt l'égalité, tantôt l'inclusion, tantôt l'appartenance. Il importe donc de traduire par des symboles différents, des

acceptations différentes du *même* mot.

Un des intérêts de la logique formelle est ainsi de dépister les ambiguïtés et de permettre l'étude des pas de raisonnement ou *inférences*, un à un, en démontrant rigoureusement leur validité sans laisser place au jeu des interprétations ou à une quelconque lacune associée à une étape non justifiée.

Après avoir été partie intégrante de la philosophie avec les grands précurseurs que furent Aristote, les stoïciens, Avicenne, Abelard, Kant, Leibniz, la logique formelle devient plus technique et plus mathématique. Avec G. Boole, J. Venn, A. de Morgan, les résultats directement utilisables dans d'autres disciplines s'élaborent et réalisent le rêve de Leibniz en construisant pratiquement une « lingua characteristica universalis ». On peut d'ailleurs ici distinguer le raisonnement formalisé de l'argumentation parlée ou écrite. A l'origine, la logique était une théorie de l'argumentation valide.

Mais c'est avec G. Frege d'abord, puis G. Peano, et enfin le monument des « Principia mathematica » de B. Russel et A. Whitehead (1913) que la logique s'affranchit complètement du langage parlé, s'érige en discipline autonome fondée sur une idéographie propre... et pose alors de nouveaux et sérieux problèmes :

- a) Vouloir étudier la logique elle-même et donc raisonner sur son formalisme ne constitue-t-il pas un cercle fondamentalement vicieux ?
- b) Dans quelle mesure peut-on faire reposer l'ensemble des mathématiques sur la logique ?
- c) Comment passer légitimement du symbolisme logique, vide de sens, à une interprétation concrète dans un domaine donné ?

Depuis les Grecs, les « Eléments » d'Euclide, en géométrie, étaient considérés comme un modèle de raisonnement déductif rigoureux. Ce n'est qu'en 1882 que M. Pasch relève quelques hypothèses tacites fondamentales (par exemple l'affirmation de l'existence d'un point entre deux autres points).

C'est David Hilbert qui propose le programme consistant à faire reposer toute la mathématique sur un ensemble totalement explicite d'axiomes dans lequel le contenu concret et conceptuel des entités manipulées ne doit pas être présupposé, mais, bien au contraire, introduit dans les axiomes eux-mêmes (1895).

Kurt Gödel montre en 1931 dans un théorème brillant, célèbre et fondamental, qu'un tel programme est impossible. Il pose ainsi les limites internes de cette théorie de la démonstration et des formalismes d'une manière générale.

L'importance d'une telle approche, qui concerne la théorie ou *metathéorie*, sera soulignée tout au long de ce chapitre et rencontrée maintes fois dans les chapitres suivants.

3.2 Définition d'un système formel

Un *système formel*, ou SF en abrégé, est un ensemble de données purement abstrait, sans liens avec l'extérieur, qui décrit les règles de manipulation d'un ensemble de symboles traités de façon uniquement syntaxique, c'est-à-dire sans considération de sens (sans sémantique).

Un SF est constitué :

- a) D'un alphabet fini de *symboles* ;
- b) D'un procédé de construction des *mots* du S.F. ;
- c) D'un ensemble *d'axiomes* qui sont des mots ;
- d) D'un ensemble fini de *règles de déduction* qui permettent de déduire d'un ensemble fini de mots un autre ensemble de mots. Elles sont de la forme :
 $U_i \text{ et } U_z \text{ et } \dots U_p \rightarrow W_1 \text{ et } W_z \text{ et } \dots W_n$
où les U_i et les W_j sont des mots du S.F.
La flèche « \rightarrow » se lit : « permet de déduire ».

3.2.1 Quelques définitions supplémentaires

Un S.F. est aussi quelquefois appelé une *axiomatique* ou une *théorie* ou encore tout simplement un ensemble de formules. L'alphabet, supposé a priori fini, est quelquefois appelé *vocabulaire*. On y distingue les constantes, les variables et les opérateurs (Cf. exemples en 3.2).

Le procédé de formation des mots (point 2) définit la construction syntaxique ou grammaire des mots (suites de symboles bien formées). Il est distinct du point 4, qui définit les déductions autorisées.

Une *preuve* (ou démonstration) est une suite finie de mots M_1, M_2, M_r dans laquelle chaque M_i est un axiome ou bien se déduit par une application d'une des règles, des mots précédents M_j , avec $j < i$

Un *théorème* est un mot t , tel qu'il existe une preuve avec $M_r ? t$.

En particulier, tout axiome est un théorème. Si t est un théorème, on écrit en abrégé :
 $\vdash t$.

Les règles (point 4) sont appelées aussi règles de dérivation ou d'inférence. Elles permettent, en principe, de distinguer les mots quelconques des théorèmes.

On a : {théorèmes} \subset {mots} \subset {chaînes de l'alphabet}
avec, dans le cas général, des inclusions strictes.

On distingue deux types de règles de déduction : d'une part, celles qui portent sur des mots considérés dans leur totalité ; elles ne s'appliquent que globalement et on parle de *productions*. D'autre part, celles qui sont autorisées à s'appliquer sur toute partie d'un mot qui est elle-même un mot du S.F. On parle alors de règles de *réécriture*.

Ainsi, en mathématique usuelle, la règle : « $x < y$ et $y < z$ entraîne $x < z$ » s'applique globalement à un mot (une proposition d'analyse) ; c'est une production (à deux antécédents) et *entraîne* est abrégé en « \rightarrow », tandis que la règle : $x - x \mapsto 0$ s'applique partout sur toute sous-expression. C'est une réécriture, ce qu'indique le signe \mapsto .

Bien sûr, règles de productions et de réécriture ne s'appliquent que dans un seul sens, de gauche à droite. Il faut que les deux règles « $a \mapsto b$ » et « $b \mapsto a$ » soient données

pour que l'on puisse réécrire dans les deux sens – c'est par exemple ce qui se passe avec les identités remarquables de notre enfance – et on doit alors prendre garde à ne pas boucler dans les démonstrations !

Note : En toute généralité, on peut noter que les ensembles des points 3 et 4 ne sont restreints qu'à être récursivement énumérables ; dans toute la suite nous supposerons cependant qu'ils sont finis.

3.2.2 Règle du jeu valable dans tout SF : règle de substitution

Pour appliquer une règle à un mot M d'un SF, on doit faire coïncider le membre gauche G de la règle avec M . Pour cela on a le droit d'effectuer des substitutions dans G et/ou dans M : une *substitution* consiste à remplacer toutes les occurrences d'une variable par un mot quelconque du SF qui ne contient pas cette variable.

3.2.3 Exemples de S.F.

Exemple numéro 1 :

- $$(JP) \left\{ \begin{array}{l} 1. \text{ alphabet } := \{a, b, \square; \} \\ 2. \text{ mots : Suite quelconque de symbole(s) } a, \text{ ou } b, \text{ ou } \square \\ 3. \text{ Axiome unique : } a \square a; \\ 4. \text{ Règle unique : } c_1 \square c_2 \rightarrow bc_1 \square bc_2 \end{array} \right.$$

Par convention, dans cette règle les symboles c_1 et c_2 désignent des suites quelconques de symbole(s) a ou b du système formel (JP), et sont substituables par n'importe quelles suites de symbole(s) a ou b .

c_1 et c_2 ne sont pas des symboles de (JP), ils ne servent que d'intermédiaires pour formaliser les règles de déduction. Par convention et à cause du procédé de formation des \square est un opérateur (non substituable) ;
mots, on dit que : a est une constante (non substituable)
 b est une constante (non substituable)

Une méthode vient immédiatement à l'esprit pour engendrer les mots valides, c'est-à-dire déductibles selon la règle 4 dans ce SF. Il suffit de considérer de façon combinatoire et exhaustive toutes les applications possibles de la règle 4 à partir de l'axiome 3. Il vient :

a	\square	a
b	a	\square b a
b	b	a \square b b a
b	b	b a \square b b b a, et caetera

Par ailleurs, de façon évidente, $baab\square abba$ ne saurait être un mot valide. Nous reviendrons sur ce SF au paragraphe 5.

Exemple numéro 2

Douglas Hofstadter, dans son merveilleux livre « Godel, Escher, Bach : an eternal golden brald », qui traite pendant quelques 700 pages de l'idée de récursion et d'autoréférence à travers la logique, les théorèmes de Godel, l'informatique, les étranges

dessins d’Escher, les fugues, canons et impromptus de Bach, introduit, dans son premier chapitre, le « MU-puzzle ». Il propose le SF suivant :

$$(JP) \left\{ \begin{array}{l} 1. \text{ alphabet : } \{M, I, U\}; \\ 2. \text{ mots : toute suite des lettres de l'alphabet ;} \\ 3. \text{ axiome unique : } MI; \\ 4. \text{ règles de déduction :} \\ \text{R1)} \quad mI \rightarrow mIU \quad (\text{production}), \\ \text{R2)} \quad Mm \rightarrow Mmm \quad (\text{production}), \\ \text{R3)} \quad III \mapsto U \quad (\text{réécriture}), \\ \text{R4)} \quad UU \mapsto \quad (\text{réécriture}) \end{array} \right.$$

Dire que R1 est une règle de production signifie qu’elle ne s’applique que si la dernière lettre d’un théorème est *I*. Dans ce cas, on peut, du théorème : « *MUIUI* », déduire le théorème : « *MIUMIUMIU* ».

Notons qu’ici encore, le symbole *m* n’appartient pas au SF. Il joue le rôle d’un mot quelconque.

Ainsi la règle R2 permet, depuis : « *MUI* » de déduire le théorème « *MUIUI* ». sous réserve, bien sûr, de savoir que « *MUI* » est un théorème...

La règle R3 autorise par exemple le passage de « *MUIIIUM* » à « *MUUUM* » .

Quant à la dernière, elle indique que toute chaîne de deux *U* consécutifs peut être purement et simplement supprimée : « *MUUUUMM* » devient « *MM* ». Faisons donc fonctionner le système (DH) :

- a) *MI* axiome
- b) *MII* règle R2 à partir de a)
- c) *MIII* règle R2 à partir de b)
- b) *MIIII* règle R1 à partir de c)
- b) *MIUU* règle R3 à partir de d) en troisième position.
- b) *MI* règle R4 à partir de e)...

Je vous laisse vous amuser maintenant : Douglas Hofstader pose la question pertinente d’obtenir, dans ce SF, le théorème *MU*. A vous !

3.3 Intérêt des systèmes formels : Décidabilité et interprétation

3.3.1 Décidabilité

La première question très naturelle qui se pose, quand on se donne un SF est de savoir s’il est facile d’inverser la mécanique : est-on capable de dire, en examinant un mot quelconque du SF, s’il est ou non démontrable, c'est-à-dire si c'est un théorème ou *un non-théorème* (mot dont on peut prouver que ce n'est pas un théorème) ?

Le rêve des mathématiciens est, qu’étant donné un SF, il existe un procédé bien déterminé qui en un nombre fini d’étapes donne la réponse. Un tel procédé, quand il existe, est appelé *procédure de décision* et le SF associé est dit *décidable*. Le problème est qu’une telle procédure, symbolisée en figure 3.1, n’existe pas toujours .

La raison essentielle pour laquelle une telle procédure n'existe pas toujours, y compris dans des théories simples et fondamentales comme le calcul des prédictats du premier ordre (Cf. paragraphe 5) est que, si les règles de dérivation appliquées de toutes les façons possibles et itérativement sur l'ensemble des axiomes permettent, par construction du SF, d'énumérer complètement les théorèmes, même s'ils sont en nombre infini, il n'existe aucun procédé semblable, en général, pour énumérer les non-théorèmes.

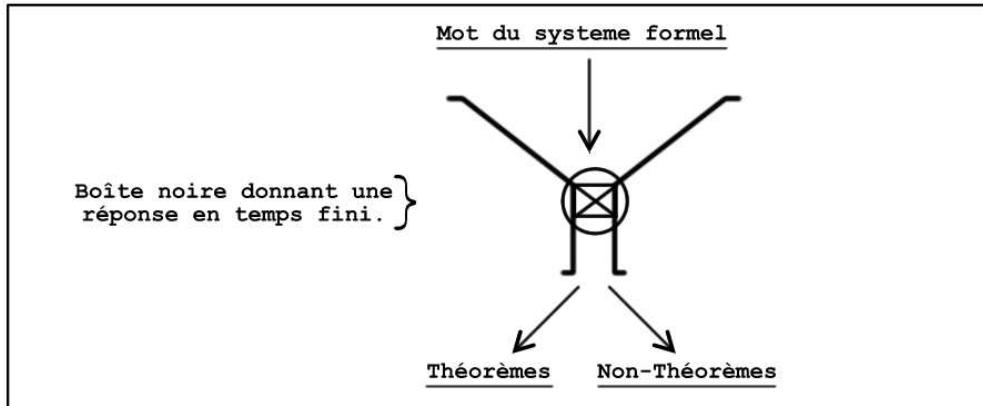
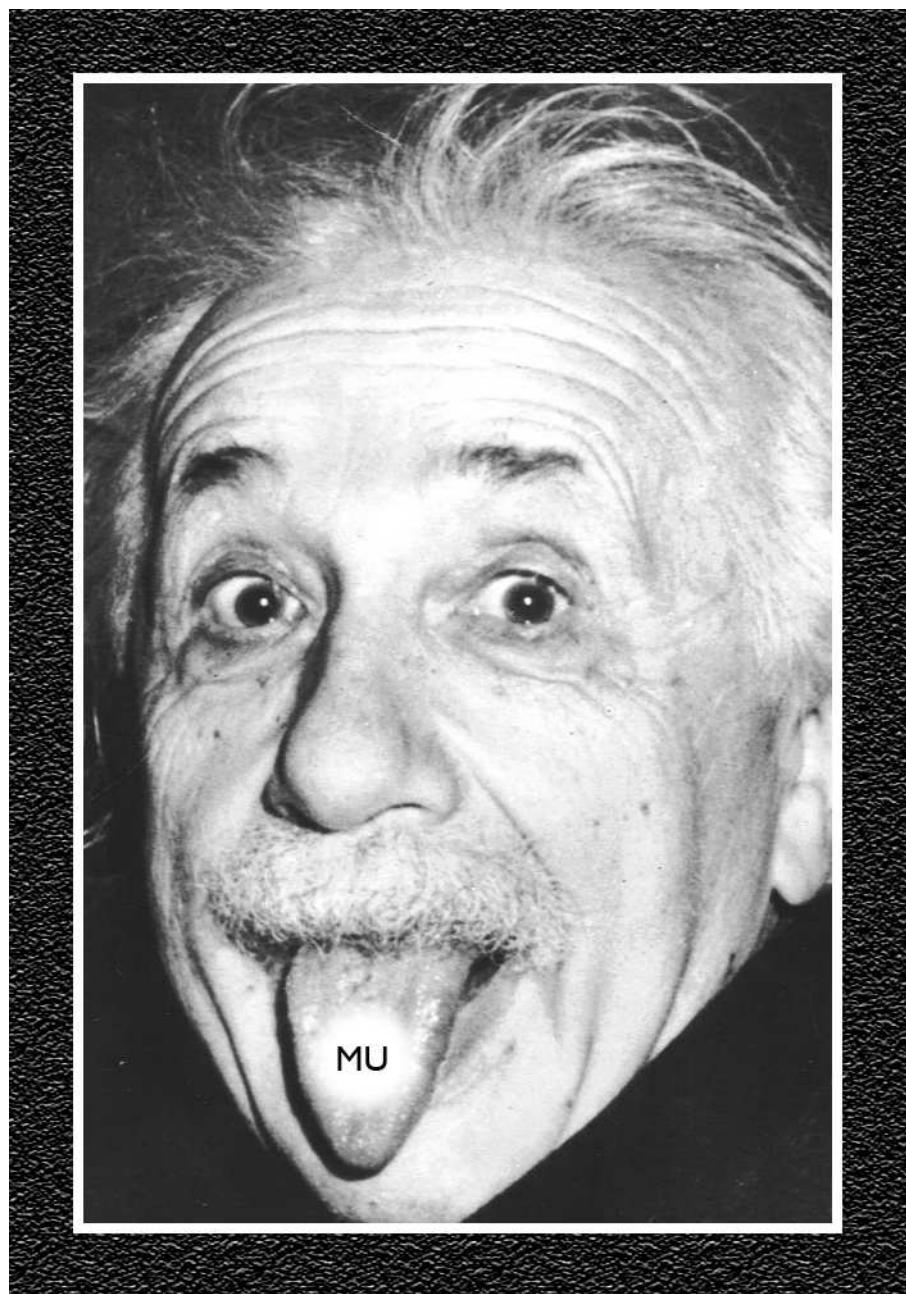


FIG. 3.1 – Décidabilité d'un système formel

La méthode combinatoire se trouve ainsi mise en défaut : si, après un certain temps, on n'obtient aucune réponse, on ne sait pas si le mot à l'entrée est un non-théorème ou bien si le SF en question n'est pas décidable. On dit que l'ensemble des théorèmes d'un SF n'est pas nécessairement *récursivement énumérable*.



3.3.2 Interprétation

Les SF ne sont pas construits gratuitement : ce sont les modélisations d'une certaine réalité concrète ou mathématique. Une *interprétation* est une mise en rapport d'un SF avec l'univers : elle donne un *sens* à tout symbole du SF en établissant une correspondance univoque entre les symboles du SF et des objets de l'univers. Les théorèmes, une fois interprétés, deviennent alors des *énoncés* au sens habituel et, en tant que tels, sont jugés *vrais* ou *faux*.

Remarquons qu'il s'agit là de **fermer la boucle** de la démarche mathématique : dans un premier temps, le mathématicien étudie la réalité en construisant une représentation abstraite, c'est-à-dire un certain SF. Il démontre ensuite des théorèmes dans ce S.F. – tout l'intérêt des SF tient d'ailleurs à cette abstraction qui fait qu'un **même système peut modéliser plusieurs situations concrètes différentes**. Dans un troisième temps, il convient de revenir au point de départ et de donner l'interprétation des théorèmes obtenus par la formalisation. Bien sûr, on s'arrange pour qu'il existe toujours, pour un SF donné, au moins une interprétation dans laquelle tout théorème du SF ait l'interprétation vraie.

Un SF sera d'autant plus intéressant qu'il existe de nombreuses interprétations où il en est ainsi : une seule démonstration fournit alors autant de résultats concrets différents.

La mathématique moderne, par la théorie des catégories et celle des modèles, s'intéresse ainsi à des SF de plus en plus généraux.

3.3.3 Démonstration et valeurs de vérité :

Selon les définitions qui précèdent, il existe, par construction, **une distinction profonde entre les concepts de preuve et de vérité** : ils appartiennent à deux univers différents. A priori, par exemple, rien n'assure que toute proposition vraie au sens commun corresponde à un mot démontrable : ce n'est pas parce que quelque chose est vrai qu'il en existe nécessairement une preuve !

En fait, quatre cas peuvent se présenter dans la relation entre démonstration et valeur de vérité : un mot est un théorème (T dans le schéma de la figure 3.2) ou est un non-théorème (NT) : son interprétation peut être jugée vraie (V) ou fausse (F).

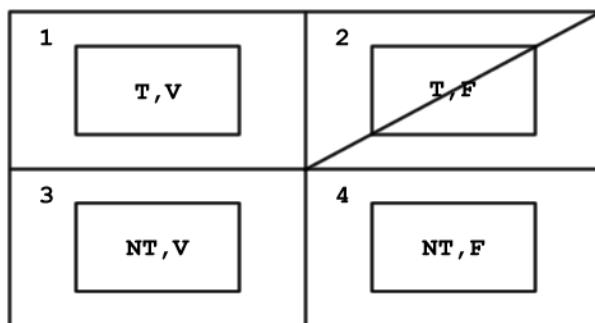


FIG. 3.2 – Démonstration et valeurs de vérité.

Les deux cas du haut, dans la figure 2, sont sympathiques : Le cas 1 dans lequel l'interprétation est vraie, est sans problème. Si un mot est démontrable et correspond,

dans une certaine interprétation, à la valeur faux, alors il est clair que l'interprétation correspondante est sans intérêt. Le cas 2 est facilement éliminé en retenant, par définition, comme correctes pour un SF donné, les interprétations qui associent à tous les théorèmes la seule valeur vraie, et seulement celles-là. Le cas des théorèmes est ainsi réglé. Mais il reste les deux cas 3 et 4 qui concernent quant à eux les non-théorèmes. Associer la valeur faux à tous les non-théorèmes semble certainement souhaitable : ce n'est pas toujours possible (Cf. exemple 3.4.1) ! Cependant les systèmes les plus utilisés sont généralement dans ce cas. Il en est par exemple ainsi de la logique des propositions présentée au paragraphe 4.

Le cas le plus revêche est alors le troisième qu'il n'est pas toujours possible d'éliminer : il peut exister des **non-théorèmes** d'un SF qui, dans certaines interprétations, sont **vrais**. Les interprétations correspondantes sont souvent retenues cependant. Ainsi, la thèse célèbre de Fermat : « *Pour tout entier n supérieur à 2, l'équation en nombres entiers : $x^n + y^n = z^n$ n'a pas de solution* » est peut-être vraie avec l'interprétation commune des symboles arithmétiques : elle a été effectivement vérifiée sur ordinateur jusqu'à des valeurs de n au-delà de la dizaine de milliers, mais elle est peut-être indémontrable dans l'axiomatisation actuelle de l'arithmétique.

Pire encore, nous verrons au paragraphe 6 qu'il existe des SF dans lesquels la classe (NT, V) n'est vide pour aucune interprétation ! Il s'agit là des premières **limitations des formalismes**. Certaines sont liées à la notion d'interprétation, d'autres en sont, en réalité, indépendantes.

Dans tous les cas, un problème immédiat se pose : comment, à partir des axiomes et à partir des règles, faire les bons choix pour aboutir rapidement à une preuve et comment démontrer qu'un mot n'est pas démontrable ? Posé ainsi, le problème est insoluble dans sa généralité. Sachant que la méthode combinatoire est dangereuse et inefficace, nous essayons nous-même de ne pas travailler en aveugle : pour être efficace nous étudions formellement chaque SF et établissons des résultats qui donnent des propriétés formelles du SF lui-même. Ce faisant, nous améliorons fortement le combinatoire. Par exemple, si un SF ne mentionne la constante A ni dans les axiomes, ni dans les règles, alors A ne peut apparaître dans aucun théorème. Autre exemple : si aucune des règles de déduction n'engendre un mot plus long (c'est-à-dire comportant un nombre supérieur de symboles) que les théorèmes auxquels elle s'applique, alors tout théorème plus long que l'axiome peut être écarté. Dans ce cas, nous sommes certains d'arriver au résultat en un temps fini puisque l'arbre de recherche est réduit à une dimension finie $\leq L^n$, où L est le nombre de symboles de l'alphabet et n la longueur du mot à démontrer.

L'importance fondamentale d'une telle *métathéorie* va maintenant être développée concrètement sur les deux exemples déjà présentés et sur deux autres SF très classiques : le calcul des propositions, puis le calcul des prédictats du premier ordre. C'est la poursuite de telles études qui conduisent vers 1930 de grands mathématiciens, notamment Church, Gödel et Tarski, à des résultats très généraux sur les SF, que nous aborderons ensuite.

3.3.4 Exemples de démonstrations

Exemple numéro 1 :

Le premier système formel (JP) possédait pour axiome et règle :

$$\begin{aligned}\vdash a \square a \\ c_1 \square c_2 \rightarrow bc_1 \square bc_2\end{aligned}$$

La méthode permet d'obtenir successivement les théorèmes suivants :

$$ba \square ba, bba \square bba, bbba \square bbba, \dots$$

On voit que les théorèmes de cette théorie coïncident exactement avec les mots de la forme :

$$\begin{array}{ccc} b \dots ba & \square & b \dots ba \\ \text{symbole } b \text{ (} p \text{ fois)} & & \text{symbole } b \text{ (} p \text{ fois)}\end{array}$$

nous pouvons décider d'abréger cela en : $bP a \square bP a$ où, par convention, bP désigne b concaténé p fois et où b^0 est le symbole vide. Cette convention introduit une métanotation puisque l'élévation à une puissance n'est pas une opération du SF initial. Notons qu'il a fallu utiliser dans le raisonnement un SF de plus haut niveau : l'arithmétique.

Note :

La nécessité de distinguer langage et métalangue apparaît dans d'autres situations, avec des difficultés connues, quand la langue naturelle est utilisée dans les deux cas :

- Indication concernant la frappe à l'intérieur d'un manuscrit.
- Définition en français de la grammaire du français.

Cette distinction est fondamentale en informatique où des programmes particuliers (compilateurs) ont pour objet de traduire d'autres programmes. La définition d'un langage de programmation sous forme Naur-Backus spécifie ainsi le SF qui donne naissance aux expressions admises de ce langage :

$$<\text{programme}> ::= <\text{instruction}> \| <\text{programme}><\text{instruction}>$$

(Un programme est une instruction ou bien un programme suivi d'une instruction).

$$\begin{aligned}<\text{instruction}> &:= * <\text{commentaire}> \| <\text{ligne}><\text{étiquette}> \\ &\quad \| <\text{étiquette}><\text{ligne}>\end{aligned}\tag{3.1}$$

$$<\text{commentaire}> ::= \text{vide} | <\text{lettre}><\text{commentaire}>\tag{3.2}$$

et ainsi de suite ; tous les symboles entre crochets sont progressivement définis.

Nous avons donc construit une *procédure de décision pour* (JP) : on compte le nombre p de symboles b successifs en tête du candidat théorème, on vérifie que le symbole suivant est un a , le suivant un \square , et qu'après avoir retrouvé p symboles b consécutifs on rencontre ensuite un a qui est le dernier symbole. Si un mot passe ce test, c'est un théorème ; c'est un non-théorème dans le cas contraire. Introduisons maintenant des interprétations sur (JP).

Première Interprétation : Convenons que le symbole a représente *zéro*, b le concept *successeur d'un nombre* sur l'ensemble des entiers et \square le signe *égalité* dans le même ensemble. L'axiome s'interprète alors comme : $0 = 0$, énoncé que nous considérons *vrai*. Les théorèmes successifs s'écrivent : $1 = 1$, $2 = 2$, $p = p$ et sont tous *vrais*. Réciproquement un énoncé comme $1 = 2$ qui ne saurait être l'interprétation d'aucun théorème (il serait issu du non-théorème : *baabba*) est pour nous, dans cette interprétation, visiblement *faux*. On a donc une interprétation correcte de (JP) dans l'arithmétique usuelle, avec correspondance parfaite $T \rightleftharpoons V$ et $NT \rightleftharpoons F$. Les classes 2 et 3 de la figure 2 sont vides.

Deuxième Interprétation : Prenons maintenant, pour a , la proposition « Socrate est mortel », pour \square , l'identité entre deux propositions et pour b , la négation. L'axiome devient sous cette interprétation :

« Socrate est mortel » est identique à « Socrate est mortel » que nous considérons comme un énoncé *vrai*.

Le premier théorème $ba\square ba$ devient : « Socrate n'est pas mortel » est identique à « Socrate n'est pas mortel » qui est encore vrai. (Remarquons que peu importe ici que Socrate soit ou non mortel, les deux énoncés précédents affirment simplement que le faux est identique au faux et le vrai au vrai).

Mais au mot $bba\square a$, qui est un non-théorème de (JP) correspond ici l'énoncé : « La négation de Socrate n'est pas mortel est identique à Socrate est mortel» qui est à nouveau vrai dans cette deuxième interprétation contrairement à ce qui se passait dans la première. Ici la classe 3 : (NT, V) n'est plus vide et le dernier énoncé ne sera jamais l'interprétation d'un théorème de (JP) !

Exemple numéro 2

Le système formel (DH) avait pour règles :

$$\begin{array}{lll} R1 & mI & \rightarrow mIU \\ R2 & Mm & \rightarrow Mmm \\ R3 & III & \rightarrow U \\ R4 & UU & \rightarrow \end{array}$$

La question est de savoir si MU est un théorème de (DH), démontrable à partir de l'axiome MI. Plaçons-nous au métaniveau et étudions formellement le système formel (DH) lui-même.

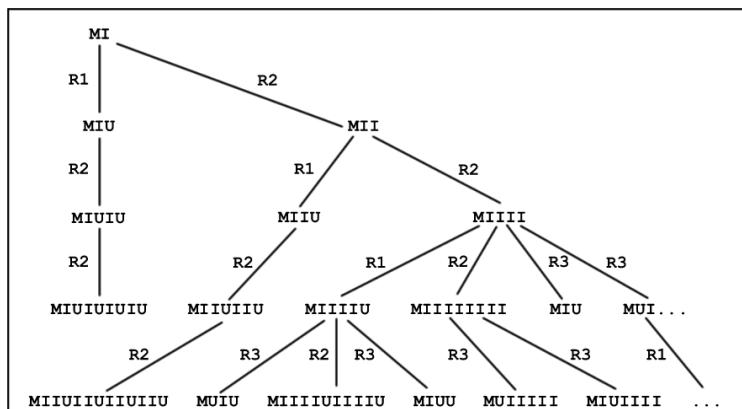


FIG. 3.3 – Arbre de dérivation pour MU

Tout d'abord il est clair que (DH) ne peut produire que des mots **commençant par le symbole M** . En effet, les deux seules productions, R1 et R2, ne changera pas le début des mots. MU est donc, de ce point de vue, un théorème possible. Par ailleurs, le but à atteindre impose de supprimer un I au moins, en partant de l'axiome. Cela ne pourra se faire que par l'action des règles R2 et/ou R3 puisque R1 et R4 laissent inchangé le nombre total de I . Mais R3 diminue le nombre total de I de 3 unités exactement. R2, quant à elle, multiplie ce nombre par un facteur 2 à chaque application. R2 fabrique donc seulement des nombres pairs de I . Tandis que R3 ne sait que diminuer ces nombres de 3 unités : pour obtenir un nombre nul de I , il faut donc appliquer R3 sur un mot comportant un nombre de I multiple de 3, et 3 exactement à la dernière application. Or, R2, qui n'engendre que des nombres pairs de I , est dans l'incapacité d'exhiber de tels mots, ainsi : **MU n'est pas un théorème du système formel (DH)**.

Notons que pour obtenir ce résultat, nous avons utilisé, à nouveau, un résultat d'un autre système formel – l'arithmétique – (et ce n'est pas un hasard), qui dit que : « pour que 3 divise $2k$, puisque 3 est premier avec 2, il faut qu'il divise k ». Ici, comme ni l'une ni l'autre des règles R2 et R3 n'est capable d'engendrer un tel multiple de 3 à partir de rien et que l'axiome ne contient pas un nombre de I multiple de 3, il est impossible d'obtenir de tels mots.

La petite étude précédente se résume bien plus simplement par un nouveau système formel qui est une modélisation des règles R2 et R3 de (DH) :

axiome	$i = 1$	(nombre de i de l'axiome)
règles	$r2) \quad i \leftarrow 2i$	(effet de la règle R2)
	$r3) \quad i \leftarrow i - 3$	(effet de la règle R3)
but :	$i = 0 ?$	(R1 et R4 laissent i inchangé)

Ce système ne pouvait fournir aucun multiple de trois I ; ($M, MUU, MIIIU, \dots$ ne sauraient donc être des théorèmes de (DH)). La figure 3.4 montre les passages entre les différentes valeurs possibles de I .

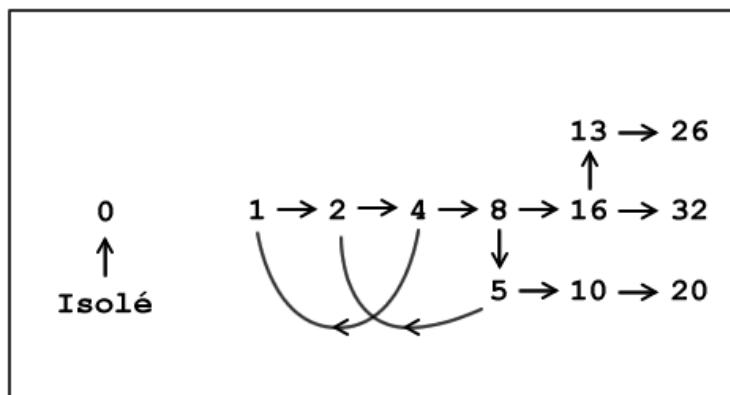


FIG. 3.4 – Une abstraction du système formel DH

Ce système formel sur I peut être simplifié à nouveau ; l'idée vient encore du schéma de démonstration (figure 3.3) : seuls les retours arrière de trois unités sont possibles. Les congruences modulo 3 permettent de traduire ce fait important.

Avec : $j = I$ modulo 3 (deuxième changement de représentation),
puisque R3 laisse j inchangé,
et que $2I$ modulo 3 $\equiv (3I - I)$ modulo 3 $\equiv -j$.

Il vient le nouveau SF sur j :

axiome :	$j = 1$
règle p3 :	$j = -j$
but :	$j = 0$

Il est ainsi évident que le but MU ne peut être atteint puisque les valeurs de j ne peuvent être alternativement que +1 et -1.

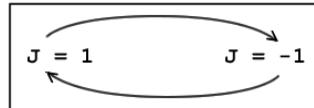


FIG. 3.5 – Abstraction du système formel sur i

Deux SF plus particulièrement utilisés, sont présentés maintenant : la logique des propositions, puis le calcul des prédicats du premier ordre.

3.4 La logique des propositions : (LP)

Il s'agit du SF donné de la façon suivante :

- a) **Alphabet :**
 - lettres propositionnelles = $p, q, r, s, t \dots$;
 - opérateurs logiques = \neg, \supset ; ils se lisent *non* et *implique* ;
 - parenthèses = $(,)$
- b) **Formation des mots :**
 - une lettre propositionnelle est un mot ;
 - si m est un mot alors (m) est un mot ;
 - si m est un mot alors $\neg m$ est un mot ;
 - si m_1 et m_2 sont des mots alors $m_1 \supset m_2$ est un mot ;
- c) **Axiomes :**
 - (A1) $(m_1 \supset (m_2 \supset m_1))$
 - (A2) $(m_1 \supset (m_2 \supset m_3)) \supset ((m_1 \supset m_2) \supset (m_1 \supset m_3))$
 - (A3) $(\neg m_2 \supset \neg m_1) \supset (m_1 \supset m_2)$
où m_1, m_2, m_3 sont des mots quelconques.
- d) **Règle de déduction unique** - dite de *détachement* ou *modus ponens* :
 - Si m_1 et $(m_1 \supset m_2)$ sont des théorèmes, alors on en déduit (m_2)
 - Soit formellement : (m_1) et $(m_1 \supset m_2) \rightarrow m_2$

L'interprétation, à la naissance même de ce SF, qui modélise le raisonnement logique habituel, considère les lettres comme des énoncés quelconques : le \neg comme la *négation*, le \supset comme l'*implication logique*. Il faut remarquer que dans (LP), ni les axiomes, ni les règles ne font jouer un rôle spécial à l'une des lettres propositionnelles. Celles-ci sont donc complètement interchangeables et si :

$\vdash (p \supset \neg q) \supset (r \supset s)$, alors, aussi bien : $\vdash (t \supset \neg s) \supset (q \supset r)$, par exemple. Tous les *renommages* des lettres propositionnelles, appelées pour cette raison *variables libres* sont autorisés. Il n'y a pas de constante dans (LP).

En outre, dans ce contexte, il est courant d'utiliser d'autres symboles : par exemple le *ET logique*, noté \wedge . Celui-ci peut en réalité être défini à partir du système précédent :

$m_1 \wedge m_2$ est équivalent, par définition, au mot $\neg(m_1 \supset \neg m_2)$.

Le *OU logique*, noté \vee est défini de même par :

$m_1 \vee m_2$ est équivalent à : $\neg(\neg m_1 \wedge \neg m_2)$

Enfin, l' *identité logique* notée \equiv , exprime que :

$m_1 \equiv m_2$ est équivalent, par définition, au mot : $(m_1 \supset m_2) \wedge (m_2 \supset m_1)$

Moyennant ces définitions, qui enrichissent (LP), le lecteur se convaincra que, dans l'interprétation indiquée, ce SF traduit les **lois familières de la pensée déductive**. En particulier, les trois *principes* d'Aristote y sont rigoureusement démontrables :

principe d'identité, soit $(p \supset p)$.

principe du tiers exclu, soit $(p \vee \neg p)$.

principe de non-contradiction, soit $\neg(p \wedge \neg p)$.

(aucun mot n'est à la fois théorème et non-théorème).

A titre d'exemple de calcul sur (LP), démontrons le premier de ces principes, l'*identité* : « Pour tout p on a $p \supset p$ ». Le mot :

$$(p \supset ((p \supset p) \supset p)) \text{ est un axiome, donc un théorème} \quad (\text{T1})$$

c'est l'axiome (A1) où l'on a pris pour mot m_2 la chaîne de symboles $(p \supset p)$ et pour m_1 la chaîne p . Or, l'axiome (A2) permet de poser comme théorème le mot :

$$((p \supset ((p \supset p) \supset p)) \supset ((p \supset (p \supset p)) \supset (p \supset p))) \quad (\text{T2})$$

avec une nouvelle fois $(p \supset p)$ pour m_2 et p pour m_1 et m_3 .

Le premier mot formé : (T1) n'est autre que la première quantité entre parenthèses du nouveau mot (T2). La règle de détachement MP affirme donc :

$$\vdash ((p \supset (p \supset p)) \supset (p \supset p)) \quad (\text{T3})$$

Or le mot $(p \supset (p \supset p))$ lui-même est un théorème, en vertu de (A1) avec m_1 et m_2 remplacés par p . La règle de détachement se déclenche donc à nouveau pour donner finalement le théorème :

$$\vdash (p \supset p) \quad (\text{T4})$$

Ce mot (T4) exprime précisément le principe d'identité.

D'autres théorèmes de base se démontrent également en quelques étapes, parmi eux on trouve :

$$p \equiv \neg \neg p;$$

$$\neg(p \vee q) \equiv (\neg p \wedge \neg q);$$

$$(p \supset q) \equiv (\neg p \vee q).$$

Le théorème $(p \supset q) \equiv (\neg p \vee q)$ indique que, pour démontrer $(p \supset q)$, il est équivalent de démontrer $(\neg p \vee q)$; c'est le principe du *raisonnement par l'absurde* (Cf. Le Principe de Résolution du paragraphe 10).

Comme dans (JP) et (DH) quelques métathéorèmes peuvent être également facilement obtenus pour (LP). Ainsi :

$$p \rightarrow \neg^{2k} p;$$

$$(p \text{ et } q) \leftrightarrow p \wedge q;$$

$$(p \supset q) \text{ et } (q \supset r) \rightarrow (p \supset r).$$

Notons dans ces derniers énoncés que le **et** est un élément métalinguistique qui n'est pas un mot de (LP). C'est le même **et** que celui qui apparaissait dans la définition du modus ponens. Ce n'est pas, en particulier, le même symbole que le \wedge de (LP) :

$(p \supset q) \text{ et } (q \supset r)$ signifie en effet : **on a les deux théorèmes** $(p \supset q) \text{ et } (q \supset r)$.

De tels métathéorèmes sont utiles, voire indispensables, pour raccourcir les démonstrations. En particulier, ils permettent d'établir de *nouvelles règles de déduction*. Donnons un exemple de démonstration d'un métathéorème : le théorème $(p \supset (q \supset r)) \supset (q \supset (p \supset r))$ est trivial dans (LP) pour les lettres propositionnelles quelconques p , q et r . Nous allons démontrer un résultat analogue mais plus fort, puisqu'il est valable pour des *mots quelconques* P , Q et R de (LP). Nous allons en fait prouver la nouvelle règle de déduction :

$$(P \supset (Q \supset R)) \longrightarrow (Q \supset (P \supset R)).$$

Admettons en effet que $(P \supset (Q \supset R))$ soit un théorème déjà démontré dans (LP). Mettons-le en correspondance avec la partie gauche de l'axiome (A2) en substituant P à m_1 , Q à m_2 , R à m_3 . Par modus ponens, on en déduit alors :

$$(P \supset Q) \supset (P \supset R).$$

Portant ce nouveau théorème, noté Y en abrégé, comme mot m_1 dans l'axiome (A1). Il vient le théorème : $(Y \supset (m \supset Y))$, où m est un mot quelconque.

Comme Y est un théorème, un nouveau modus ponens permet d'en déduire $(m \supset Y)$, soit :

$$(m \supset ((P \supset Q) \supset (P \supset R))).$$

Substituons encore dans (A2) m_1 par m , m_2 par $(P \supset Q)$ et m_3 par $(P \supset R)$. Il vient par un troisième modus ponens : $(m \supset (P \supset Q)) \supset (m \supset (P \supset R))$.

Remplacer le mot m par le mot Q dans ce résultat redonne (A1) en partie gauche. Un dernier modus ponens permet finalement d'en déduire :

$$(Q \supset (P \supset R))$$

on a bien ainsi démontré une nouvelle règle de déduction ou *métathéorème* dans (LP), c'est-à-dire une nouvelle méthode pour trouver des théorèmes :

$$(P \supset (Q \supset R)) \rightarrow (Q \supset (P \supset R))$$

avec P , Q et R mots quelconques de (LP).

Remarque importante sur les notations usuelles :

Par abus de langage, nous écrivons tous, en abrégé, les théorèmes sous la forme :

$H \Rightarrow C$ (hypothèses **impliquent** conclusion)

au lieu de la forme correcte :

$$(H \text{ et } H \supset C) \rightarrow C.$$

Le symbole \Rightarrow usuel est une sorte de concaténation du \supset de (LP) et du \Rightarrow de meta LP, qui **inclus** déjà le modus ponens.

De nombreux travaux ont été effectués sur (LP) depuis Aristote. En 1910 Whitehead et Russell donnaient enfin une présentation rigoureuse de (LP) et établissaient de nombreuses démonstrations. Curieusement, dans leurs travaux, (LP) comportait non pas 3 mais 4 axiomes... Ce n'est que vingt années plus tard, que J. Lukasiewicz montra que l'un d'entre eux était redondant puisque démontrable à partir des trois autres. C'est ce SF réduit que nous avons présenté. Par souci d'économie, (LP) peut être construit à partir d'un seul opérateur au lieu de deux : la barre « | » de H.M. Sheffer définie par :

$$p | q \equiv (\neg p) \vee (\neg q).$$

Le symbole « | » s'interprète comme *l'incompatibilité*. J. Nicot, à partir de cette seule connective a montré en 1917 que (LP) pouvait reposer sur un *axiome unique* :

$$(p | (q | r)) | (s | (s | s)) | ((t | q) | ((p | t) | (p | t))).$$

On prouve en effet que cet axiome unique avec son opérateur unique engendre exactement les mêmes théorèmes que le système formel original (LP).

3.4.1 Interprétation de la logique des propositions

En toute rigueur et généralité, une interprétation de (LP) peut être définie à partir de l'ensemble B à deux éléments, soit :

$B = \{0, 1\}$, où 1 et 0 jouent respectivement les rôles de FAUX et de VRAI, et de deux fonctions :

c (comme complémentaire) de B dans B , définie par $c(x) = 1 - x$;

v (comme vrai) de $B \times B$ dans B , avec $v(x, y) = 1 \iff (x = 0 \text{ et } y = 1)$.

Soit, maintenant, ϕ une application de l'ensemble des formules propositionnelles dans B , à laquelle on impose les deux propriétés suivantes :

$$\phi(\neg p) = c(\phi(p))$$

$$\phi(p \supset q) = \vee(\phi(p), \phi(q))$$

Cette deuxième égalité montre, qu'ici, peut-être contrairement à l'intuition, si $\phi(p)$ vaut 1 et $\phi(q)$ vaut 0, alors $\phi(p \supset q) = v(1, 0)$ est égal à 0. Autrement dit :

FAUX \supset *VRAI* est *VRAI* et le faux implique le vrai.

On dit qu'un mot m de (LP) est *valide*, ou encore qu'il est une *tautologie*, si et seulement si, pour toute application vérifiant les deux propriétés énoncées, on a :

$$\phi(m) = 0$$

Il est facile de vérifier que les trois axiomes de (LP) sont des tautologies. Faisons cet exercice pour (A1) :

$$\begin{aligned} \phi(m_1 \supset (m_2 \supset m_1)) &= v(\phi(m_1), \phi(m_1 \supset m_1)). \\ &= v(\phi(m_1), v(\phi(m_2), \phi(m_1))). \end{aligned}$$

Donc, avec $m \equiv (m_1 \supset (m_2 \supset m_1))$:

$\phi(m) = 0$ si $\phi(m_1) = 1$ ou bien si $v(\phi(m_2), \phi(m_1)) = 0$.

Si $\phi(m_1) = 1$, alors $\phi(m)$ vaut nécessairement 0. Si maintenant $\phi(m_1) = 0$ alors, par définition de v , $v(\phi(m_2), 0)$ ne peut être égal à 1. Ainsi, **pour toute interprétation** ϕ , l'axiome (A1) reçoit la valeur 0 (VRAI) et constitue donc une tautologie.

De plus, si (m_1) et $(m_1 \supset m_2)$ sont des tautologies, alors puisque : $\phi(m_1 \supset m_2) = v(\phi(m_1), \phi(m_2)) = v(0, \phi(m_2)) = 0$ par hypothèse, et que $\phi(m_2)$ ne peut valoir 1, m_2 est aussi une tautologie. Ceci montre la cohérence de la règle de détachement avec toutes les interprétations. En continuant, par récurrence sur la longueur des mots, le raisonnement précédent, on parvient au métathéorème 1 :

(MT1) : Tout théorème de la logique des propositions est une tautologie.

La classe d'interprétation définie ci-dessus rend le système (LP) *isomorphe* à l'algèbre de Boole classique avec ses deux lois \vee et \wedge et la complémentation. Cette considération permet d'établir l'important métathéorème 2 :

(MT2) : Toute tautologie est un théorème de la logique des propositions.

(MT1) et (MT2) ont donc pour conséquence immédiate la *décidabilité de la logique des propositions*.

3.4.2 Décidabilité de la logique des propositions

Emile Post démontre ainsi en 1921 le résultat suivant :

« Une formule F est démontrable dans (LP) si, et seulement si, elle est valide, c'est-à-dire vraie dans toutes les interprétations de (LP). »

La **logique des propositions** est donc :

- *non contradictoire* : t et $\neg t$ n'y sont pas simultanément dérivables ;
- *complète* : Les théorèmes coïncident exactement avec les mots valides.
- *résoluble* (ou *décidable*) :

Il existe une procédure de décision. En effet, les **théorèmes de (LP) coïncident avec les tautologies**, c'est-à-dire les propositions dont la valeur de vérité est VRAI quelles que soient les *valeurs de vérités* VRAI (0) ou FAUX (1) de toutes les variables propositionnelles sur lesquelles elles portent. Les *tables de vérité* des opérateurs élémentaires sont définies en accord avec l'interprétation intuitive. Ainsi :

$$\neg \quad \begin{array}{|c|c|c|} \hline p & 1 & 0 \\ \hline \neg p & 0 & 1 \\ \hline \end{array}$$

$$\supset \quad \begin{array}{|c|c|c|c|c|} \hline p & 1 & 1 & 0 & 0 \\ \hline q & 1 & 0 & 1 & 0 \\ \hline p \supset q & 0 & 0 & 1 & 0 \\ \hline \end{array}$$

Dans l'interprétation logique ainsi définie, on voit que $p \supset q$ n'est « faux » que lorsque p est vrai et q faux. Cette expression « $p \supset q$ » est effectivement équivalente à $\neg p \vee q$. L'interprétation habituelle du symbole \supset qui est *implique* est, dès lors, assez malheureuse puisque $(p \supset q)$ et $(\neg p \vee q)$ sont vraies dès que p est faux (ce qui se traduit

quelquefois par : « le faux implique n'importe quoi »). Or, cet énoncé correspond mal au sens de *implique* dans le langage courant et cette mauvaise interprétation crée quelques troubles.

La méthode bien connue des tables de vérité est prouvée par Post en 1932 : « *Pour montrer qu'un mot de (LP) est un théorème, il faut et il suffit qu'il prenne la valeur d'interprétation 0, quelles que soient les valeurs des variables propositionnelles qu'il contient* ».

Exemple :

p	q	r	$q \supset r$	$p \supset q$	$p \supset r$	$(q \supset r) \supset (p \supset r)$	$(p \supset q) \supset ((q \supset r) \supset (p \supset r))$
0	0	0	0	0	0	0	0
0	0	1	1	0	1	0	0
0	1	0	0	1	0	0	0
0	1	1	0	1	1	1	0
1	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0
1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0

TAB. 3.1 – Preuve de $(p \supset q) \supset ((q \supset r) \supset (p \supset r))$ par la méthode des tables de vérité.

Cette coïncidence entre théorèmes et *interprétations vraies* n'est pas le cas général : le système (JP) en montrait un contre-exemple.

Par ailleurs, le résultat fondamental : « **La logique des propositions est décidable** » est un théorème d'un métasystème qu'il faudrait élaborer.

Pour l'établir, nous devons utiliser un raisonnement par récurrence qu'il conviendrait de formaliser. Cette suite des métaméta-. . .-SF ne sera pas construite ici.

Le résultat de Post est étonnant car il traduit l'adéquation parfaite de (LP) avec le raisonnement classique et le langage habituel. Il affirme que la pensée et son discours sont, au moins en partie, formalisables et réductibles à des schémas purement symboliques. C'était justement ici que l'on s'attendait aux plus grandes difficultés dans le programme de formalisation des mathématiques de Hilbert (1885). Or, alors que cette étape fut franchie avec rigueur et élégance, c'est plus loin, à propos de la théorie des nombres que surgirent les difficultés.

3.5 Le calcul des prédictats du premier ordre

Le tout premier exemple mentionné : « Les hommes sont mortels... » ne peut être traduit dans (LP). En effet, l'énoncé en question fait intervenir une *variable quantifiée* « homme » et le pluriel dans cette première prémissse indique *l'universalité* :

« **Tout** x qui a la propriété homme est mortel ».

Pour traduire cela, le SF sous-jacent doit être plus riche que (LP). Les propositions p, q, r, \dots ne peuvent plus être considérées comme des entités mais doivent mentionner elles-mêmes des paramètres libres dits *variables d'individus*. La première prémissse de

l'exemple s'écrit :

$$(\forall x) (\text{homme}(x) \supset \text{mortel}(x)).$$

où x est une variable quantifiée ou liée ; « homme » et « mortel » sont des prédictats. Au premier ordre, ces prédictats eux-mêmes, ne sont pas quantifiés. Le SF qui correspond à ce point de vue est appelé *calcul des prédictats du premier ordre*, (PP) en abrégé, et est défini ainsi :

a) **Alphabet :**

- *constantes* a, b, c, d, e, ... ;
- *variables d'individus* : t, u, v, w, x, y, z, ... ;
- *prédictats* : A, B, C, D, ... ;
- *opérateurs logiques* : \neg , \supset ;
- *quantificateur* : \forall (il se lit : « quel que soit »).

b) **Formation des mots** : semblable à (LP), mais, en outre :

on affecte à chaque prédictat un poids k ; alors
 $A(x_1, x_2, \dots, x_k)$ est un mot, si et seulement si
le poids de \wedge est k .

$((\forall x_1) A(x_1, x_2, \dots, x_k))$ est un mot dans lequel
 x_1 est dite *variable liée* et
 x_i est dite *variable libre* si $i \geq 2$.

c) **Axiomes** : les trois axiomes de (LP) sont repris, mais on ajoute les axiomes (A4) et (A5) :

- (A4) $((\forall t) B(t)) \supset B(u)$ (« *particularisation* »)
- (A5) $((\forall t) (m_1 \supset m_2)) \supset (m_1 \supset (\forall t) m_2)$,
où m_1 et m_2 sont des mots quelconques,
et t n'est pas une variable libre de m_1 .

d) **Règle de dérivation** :

(m_1) et $(m_1 \supset m_2) \rightarrow m_2$	(modus ponens)
$m_1 \rightarrow (\forall t) m_1$	(généralisation)

où t est variable libre dans m_1 .

L'axiome (A4) joue un rôle fondamental, on l'appelle aussi axiome de *spécification* et on dit que, dans (A4), la variable t est *instanciée* en la variable u . La variable u devient alors une *variable libre*. Comme dans (LP), l'introduction des symboles \vee , \wedge et \equiv résulte de définitions supplémentaires. En outre, le quantificateur \exists (*il existe*) est lié au \forall par l'identité :

$$(\exists x) B(x) \equiv \neg((\forall x) \neg B(x))$$

- Tout SF qui comporte les axiomes de (PP) est dit du premier ordre : de nombreuses théories usuelles sont des variantes de (PP) par ajout d'un ou plusieurs axiomes et/ou règles de dérivation.
- Le qualificatif « premier ordre » tient au fait que dans (PP) seules les variables prédictives, symboles les plus internes des formules, peuvent être quantifiées. À l'ordre deux, c'est sur les prédictats eux-mêmes que peuvent porter les symboles \exists et \forall . Dans les ordres supérieurs, les quantifications agissent sur des prédictats de prédictats et ainsi de suite.

Exemple de démonstration dans (PP) :

Quel que soit le prédictat D de poids deux, on a :

$$(\forall y) (\forall x) D(x, y) \supset (\forall x) (\forall y) D(x, y).$$

En effet, selon l'axiome (A4) de particularisation, avec y pour t , il vient, en partant du membre gauche du théorème à démontrer :

$$(\forall y) (\forall x) D(x, y) \supset (\forall x) D(x, y).$$

à nouveau, avec x pour t :

$$(\forall x) D(x, y) \supset D(x, y)$$

La dernière règle de dérivation permet alors d'obtenir :

$$D(x, y) \supset (\forall y) D(x, y); \text{ puis } : (\forall y) D(x, y) \supset (\forall x) (\forall y) D(x, y).$$

Comme le métathéorème : $(A \supset B) \text{ et } (B \supset C) \rightarrow (A \supset C)$ est démontrable tant pour (LP) que pour (PP), il s'ensuit enfin, avec $B \equiv D(x, y)$, que :

$$(\forall y) (\forall x) D(x, y) \supset (\forall x) (\forall y) D(x, y) \quad (\text{cqfd.})$$

On démontre de façon semblable d'autres théorèmes importants de (PP) :

$$\begin{aligned} & (\exists y) (\forall x) D(x, y) \supset (\forall x) (\exists y) D(x, y); \\ & \neg((\forall x) (\forall y) D(x, y)) \equiv (\exists x) (\exists y) (\neg D(x, y)); \\ & \neg((\exists x) (\exists y) D(x, y)) \equiv (\forall x) (\forall y) (\neg D(x, y)); \\ & (\forall x) D(x) \supset (\exists x) D(x). \end{aligned}$$

Premier théorème de Gödel : « complétude de (PP) »

En 1930. Kurt Godel démontre le résultat suivant :

« Dans le calcul des prédictats du premier ordre, les théorèmes coïncident avec les formules **logiquement valides**, c'est-à-dire celles qui sont vraies dans toutes les interprétations ».

Ce théorème est le pendant, dans (PP), du résultat de Post pour (LP). Mais malheureusement, il ne conduit pas, contrairement à ce qui se passait pour (LP), à une procédure effective de décision. Le paragraphe 6.3 précisera ce point.

3.5.1 L'arithmétique formelle

Introduisons le SF particulier *théorie des nombres*, ou *arithmétique formelle* (AF), dû à Giuseppe Peano, qui est une extension de (PP). Aux cinq axiomes de (PP) sont adjoints neuf autres axiomes qui vont intervenir :

- une constante unique : 0 (*zéro*).
- quatre opérateurs : σ (*successeur*) de poids un :
 - $\sigma(0)$ est noté 1 et $\sigma(n)$ noté $(n + 1)$.
 - : + (*plus*).
 - : \times (*multiplié par*).
 - : = (*égale*); cet opérateur joue un rôle spécial dans les nouveaux axiomes.

Ces trois derniers opérateurs sont tous de poids 2.

Les neuf nouveaux axiomes s'écrivent, x , y , et z désignant toute suite de symboles de (AF) ne contenant pas le signe $=$, sous la forme suivante :

(A6)	$(\forall x) \quad x + 0 = x$
(A7)	$(\forall x) \quad x \times 0 = 0$
(A8)	$(\forall x) \quad \neg(\sigma(x) = 0)$
(A9)	$(\forall x) \quad (\forall y) \quad x + \sigma(y) = \sigma(x + y)$
(A10)	$(\forall x) \quad (\forall y) \quad x \times \sigma(y) = x \times y + x$
(A11)	$(\forall x) \quad (\forall y) \quad (\sigma(x) = \sigma(y)) \supset (x = y)$
(A12)	$(\forall x) \quad (\forall y) \quad (x = y) \supset (\sigma(x) = \sigma(y))$
(A13)	$(\forall x) \quad (\forall y) \quad (\forall z) \quad (x = y) \supset ((x = z) \supset (y = z))$
(A14)	$(A(0) \text{ et } (\forall u) \quad (A(u) \supset A(\sigma(u))) \supset (\forall u) \quad A(u))$ pour tout mot $A(u)$ du système.

Le dernier axiome formalise le *raisonnement par récurrence* appelé aussi *induction formelle*. Notons qu'un certain nombre de résultats très classiques ne sont pas donnés dans cette axiomatique mais doivent être redémontrés.

Par exemple : $(x = y) \supset (y = x)$, ou bien également : $x + y = y + x$.

En réalité, ce SF de l'arithmétique formelle (AF) est d'une **importance extrême**. Nous l'avons déjà pressenti, puisque c'est en faisant appel à (AF) que nous avons pu écrire les métathéorèmes utilisés dans (JP) et (DH). Etablissons plus nettement les rapports de ces deux systèmes avec l'arithmétique. Rappelons la donnée de (JP) :

$$(JP) \quad \begin{cases} \text{axiome (A)} : a \square a \\ \text{règle R} : m_1 \square m_2 \rightarrow bm_1 \square bm_2 \end{cases}$$

où m_1 et m_2 sont des chaînes quelconques de caractères a et b.

Les symboles de ce SF, soit a , b , \square , ne sont que des *codes* qui peuvent être remplacés par n'importe quels autres codes, par exemple par 1, 2 et 3. L'axiome (A) devient : « 1 3 1 » et la règle R dit que : « Si m_1 , suivi de 3, suivi de m_2 est un théorème, alors le mot 2, suivi de m_1 , suivi de 3, suivi de 2, suivi de m_2 est un théorème ».

Mais l'opération **être suivi de**, qui porte sur des suites de symboles, possède une correspondance dans (AF). En effet, dans l'arithmétique usuelle, pour former le nombre x suivi du nombre y , en utilisant la notation décimale, il suffit de multiplier x par une puissance de 10 convenable et d'ajouter y . Plus précisément, si y comporte n chiffres, c'est-à-dire s'écrit sous la forme :

$$\sum_{i=0}^{n-1} 10^i \times c(i)$$

où le nombre $c(n - 1)$ n'est pas nul, le nombre : « x suivi de y » n'est autre que le nombre : $10^n \times x + y$. Pour exprimer la conclusion de la règle R, on part de la droite de celle-ci :

$$m_2 = \sum_{i=0}^{n_2-1} 10^i \times d(i)$$

on forme le nombre $10^{n_2} \times 2$ et on ajoute m_2 . On obtient ainsi un nombre de $n_2 + 1$ chiffres.

Ensuite, pour former 3 suivi de 2 m_2 , on ajoute $3 \times 10^{n_2+1}$ au nombre précédent.

Pour obtenir 2 suivi de m_1 , à partir de $m_1 = \sum_{i=0}^{n_1-1} 10^i \times e(i)$, on ajoute de même $10^{n_1} \times 2$ à m_1 .

Enfin, pour obtenir « 2 m_1 3 2 m_2 », on ajoute $(2 \times 10^{n_1} + m_1)$ multiplié par 10^{n_2+2} au nombre $3 \times 10^{n_2+1} + 2 \times 10^{n_2} + m_2$. Au total, la **règle R peut être exprimée sous forme complètement arithmétisée**. Elle devient la règle (RA) :

$$(RA) \left\{ \begin{array}{l} \text{Si on a le théorème :} \\ m_1 \times 10^{n_2+1} + 3 \times 10^{n_2} + m_2, \text{ où les nombres } m_1 \text{ et } m_2 \text{ comportent} \\ \text{respectivement } n_1 \text{ et } n_2 \text{ chiffres, soit :} \\ m_1 = \sum_{i=1}^{n_1-1} 10^i \times e(i) \text{ et } m_2 = \sum_{i=1}^{n_2-1} 10^i \times d(i) \\ \text{alors on a le théorème :} \\ (2 \times 10^{n_1} + m_1) \times 10^{n_2+2} + 3 \times 10^{n_2+1} + 2 \times 10^{n_2} + m_2. \end{array} \right.$$

Ainsi, à partir du théorème : 2213221 avec $n_1 = n_2 = 3$, on engendre par (RA) le théorème :

$$\begin{aligned} & (2 \times 10^3 + 221) \times 10^5 + 3 \times 10^4 + 2 \times 10^3 + 221 \\ \text{soit : } & 222100000 + 30000 + 2000 + 221 \\ \text{soit encore } & 222132221. \end{aligned}$$

L'étude du système (JP) est ainsi complètement ramenée à celle d'un sous-système de (AF).

Nous pouvons procéder de même pour le système (DH). Codant les symboles M , I et U respectivement par 2, 1 et 0, l'axiome devient : 2 1, et la règle (R3), par exemple, ($III \mapsto U$) s'écrit :

« Si $m_1 \times 10^{n_2+3} + 111 \times 10^{n_2} + m_2$ est un théorème (m_2 possédant n_2 chiffres), alors on a aussi le théorème : $m_1 \times 10^{n_2+1} + m_2$ »

Le système (DH) est ainsi également plongé dans (AF).

La démonstration de théorèmes et de métathéorèmes dans (JP) ou (DH) se ramène donc aux démonstrations dans (AF). Bien sûr, à ce stade, il est possible de donner pour chaque SF arithmétisé dans (AF), une *interprétation dans (AF) lui-même*. Ainsi, dans (JP) arithmétisé, posons :

- « 1 » interprété en « 1 »;
- « 2 » interprété en « successeur »;
- « 3 » interprété en « = ».

Nous retrouvons à partir du théorème : 222132221, l'énoncé « vrai » au sens habituel : « 4 = 4 ». La boucle est ainsi bouclée. Ce type de démarche constitue la première des idées qui va finalement mener au deuxième théorème de Kurt Gödel.

3.5.2 Deuxième théorème de Gödel (1931) : Incomplétude de l'arithmétique.

L'arithmétisation peut être effectuée pour tout système formel. En particulier, il est possible de coder par l'arithmétique les systèmes (LP) et (PP) et... (AF) lui-même n'échappe pas à la règle ! La preuve de Kurt Gödel va s'inspirer de ce curieux automorphisme, en établissant une variante dans (AF) du « paradoxe » du menteur, dû

à Eubulide, dialecticien de l'école de Mégare (4^{ème} siècle), fondée par le philosophe Euclide Le Socratique. L'énoncé de ce paradoxe est :

« Epiménide dit : « Les Crétois sont menteurs »;
or Epiménide lui-même est Crétois »...

Ce genre de proposition se posait comme une critique de la logique aristotélicienne. Elle échappe en effet au moule du syllogisme et fut très discutée dans la Grèce antique par les stoïciens et leurs commentateurs. En réalité cette phrase n'est pas du tout paradoxale, puisque dans l'hypothèse où Epiménide est effectivement menteur, ce qu'il dit est faux. En prenant la négation de ce qu'il dit, il vient : « Il existe un Crétois qui ne ment pas », ce que rien ne contredit. (L'erreur classique est de mal exprimer la négation et d'en déduire le « paradoxe »).

La logique connaîtra ensuite une longue période sans que s'ajoutent de résultats fondamentalement nouveaux et il faut attendre Leibniz, puis les logiciens modernes, pour que les travaux des anciens soient repris, systématisés et plongés dans l'ensemble de la mathématique. Tout ceci pour aboutir au bout du compte au second théorème de Gödel : Il existe dans AF des mots m , tels que ni m , ni $\neg m$, n'est démontrable.

Nous suivrons pour la démonstration de ce théorème la présentation de Jean Ladrière dans « Les limitations internes des formalismes » (Nauwolaerts et Gauthier Villars 1957), ainsi que celle de Doug Hofstadter dans GEB. Mais ouvrons d'abord une parenthèse pour rappeler un résultat alors récent, dont Kurt Gödel s'est assurément inspiré. Il s'agit du procédé « diagonal » de Georges Cantor, qui montre, en 1873, que « l'ensemble des nombres réels \mathbb{R} ne peut être dénombré au moyen des nombres entiers \mathbb{N} ». Pour cette preuve, Cantor ne considère que les réels compris entre 0 et 1. Son idée est alors de construire la liste (infinie) L de ces nombres dans la représentation décimale et de supposer que cette liste L peut être numérotée. La liste L se présente alors ainsi :

0, $a_1 \ b_1 \ c_1 \ d_1 \dots$
0, $a_2 \ b_2 \ c_2 \ d_2 \dots$
.....
0, $a_n \ b_n \ c_n \ d_n \dots$
.....
.....

FIG. 3.6 – La liste de Cantor

Formons alors un nombre, $\delta(L)$, du même type, mais qui ne fait sûrement pas partie de la liste : il suffit pour cela de prendre pour $k^{\text{ième}}$ décimale de $\delta(L)$ la $k^{\text{ième}}$ décimale du $k^{\text{ième}}$ nombre de la liste L augmentée d'une unité. On obtient ainsi le réel :

$$0, (a_1 + 1)(b_2 + 1)(c_3 + 1) \dots (k_k + 1) \dots$$

qui, par construction, et pour une décimale au moins, est différent de tous les nombres de L . Cela prouve qu'une numérotation, au moyen de \mathbb{N} , des nombres réels, ne peut exister. Retenons l'idée de la preuve qui est de faire jouer **au même entier k deux rôles** simultanés. Dans l'argument précédent, k désigne en effet **à la fois l'indice** (vertical) d'un réel dans la liste L et la **position** (horizontale) d'un chiffre dans un développement. De cet usage « diagonal » de k naît la preuve de Cantor. Il existe un autre paradoxe qui est en étroite relation avec la méthode diagonale de G. Cantor et qui nous rapprochera encore de la démarche de K. Gödel, c'est le :

Paradoxe de J. Richard (1907)

Ce paradoxe ne repose plus sur la notion de vérité comme chez Eubulide mais sur celle de définition. Considérons les définitions en français des nombres naturels ; chaque nombre peut être décrit de façon univoque par un ensemble fini de symboles : lettres, chiffres, signes de ponctuation. Plusieurs définitions peuvent évidemment caractériser le même nombre. Convenons alors de ne retenir, pour un nombre donné, que la **plus courte** d'entre elles, c'est-à-dire celle comportant le plus petit nombre de symboles.

Intéressons-nous maintenant au sous-ensemble C des entiers définis en moins de cent symboles. Ce sous-ensemble C est fini puisqu'il n'y a qu'un nombre fini de permutations de l'ensemble fini des symboles. Il existe par conséquent, puisque \mathbb{N} est infini, des entiers qui n'appartiennent pas à C . Soit r le plus petit d'entre eux. On peut définir r comme : « le plus petit nombre naturel qu'on ne peut définir en moins de cent symboles ». Mais précisément, cette caractérisation de r possède moins de cent symboles, d'où une contradiction et le paradoxe lié à la circularité de notre définition...

La première étape de la démonstration de Gödel est de coder effectivement sous forme de nombres tous les énoncés de (AF). La construction d'un énoncé « paradoxal » aboutira finalement à son célèbre deuxième théorème : « Le système (AF) contient une expression e^* telle que ni e^* ni $\neg e^*$ ne sont démontrables dans (AF) ».

3.5.3 Mécanisme de la preuve de Gödel

L'idée est de construire une expression qui affirme sa propre indémontrabilité. Ceci s'effectue en trois temps :

- Mise en correspondance du système formel de l'arithmétique (AF) avec les entiers (*Gödélisation*) ;
- Construction d'une propriété particulière $A(x)$ dont on ne sait si c'est ou non un théorème de (AF) ;
- Substitution, dans $A(x)$, de x par l'entier associé à $A(x)$ elle-même.

Etudions ces points un à un.

a) Gödélisation du système AF : le système (AFG)

Le système (AF) peut être arithmétisé : tout théorème de (AF) peut être codé sous forme de nombre. Mais, dès lors, tout nombre est aussi un théorème, et tout théorème peut être compris de deux façons différentes :

- comme un théorème de (AF),
- comme un théorème sur les théorèmes de (AF), autrement dit comme un métathéorème correspondant à la démonstration d'un théorème.

Le système (AF) contient ainsi son propre métasystème.

Rendons tout cela un peu plus concret :

- 1) Comme nous l'avons fait dans d'autres cas, nous savons que nous pouvons associer à tout symbole u de (AF) un codage particulier, appelé ici *nombre de Gödel*, noté $G(u)$.

- 2) A chaque suite de symboles U nous associons de même son nombre de Gödel $G(U)$, à l'aide d'une fonction de composition f , soit : $G(U) = f(G(u_1), \dots, G(u_q))$, où (u_i) est la suite des symboles qui constituent U .
- 3) En particulier, à toute *démonstration* Δ , suite d'axiomes et de règles avec substitutions, nous sommes à même d'associer le nombre : $G(\Delta) = f(G(U_j))$ où (U_j) est la suite des théorèmes utilisés dans la démonstration Δ .

Ainsi, à toute démonstration Δ dans (AF) correspond un nombre, son nombre de Gödel $G(\Delta)$. Tout raisonnement sur (AF) se traduit par un calcul sur l'ensemble \mathbb{N} des entiers. Au lieu de manipuler des symboles, des théorèmes, des démonstrations, on se contente alors de **calculs sur des entiers**. A tout énoncé tel que : « t est démontrable dans (AF) » correspond un nombre déterminé que nous noterons $Dem(t)$.

Finalement, cela revient bien à dire :

Le système formel métal-(AF) est contenu dans \mathbb{N} , lui-même contenu dans une interprétation de (AF).

Soit :

$$\text{métal-(AF)} \supset \mathbb{N} \supset (\text{AF}).$$

(AF) est ainsi dans la situation d'une langue naturelle que rien n'empêche d'être utilisée pour parler d'elle-même.

Un choix convenable de f permet de rendre **injectif** le passage de Δ à $G(\Delta)$, c'est-à-dire de faire correspondre deux nombres différents à deux démonstrations différentes. On peut choisir par exemple de faire correspondre à chaque symbole de l'alphabet un nombre premier, suivant la table :

u	$=$	0	\times	$+$	1	σ	\supset	\neg	\vee	\wedge	\forall	\exists	$($	$)$	\equiv	x	y	\dots
$G(u)$	2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53	59	\dots

Chaque formule U , composée de q symboles (u_i) , pour i de 1 à q , est à son tour codée, à l'aide de la suite des q premiers nombres premiers, par le nombre :

$$G(U) = 2^{G(u_1)} \times 3^{G(u_2)} \times 5^{G(u_3)} \times \dots \times Pq^{G(u_q)}.$$

où Pq est le q^{ime} nombre premier.

Une démonstration, c'est-à-dire une suite de r formules U_1, U_2, \dots, U_r , sera codée de la même façon par :

$$G(\Delta) = 2^{G(U_1)} \times 3^{G(U_2)} \times 5^{G(U_3)} \times \dots \times P_r^{G(U_r)}$$

Inversement, grâce à ce procédé, il est possible à partir d'un nombre quelconque, par **décomposition en facteurs premiers** (décomposition que nous savons unique en vertu du théorème fondamental de l'arithmétique) de revenir en deux temps aux puissances $G(ui)$, c'est-à-dire aux symboles primitifs de (AF). Ce jeu n'est que purement théorique car les $G(\Delta)$ deviennent vite trop grands pour être manipulables ; mais l'essentiel est qu'il soit simplement possible.

Exemple

Ainsi, étant donné un nombre T , éventuellement très grand, censé représenter une démonstration, il est décomposable en facteurs premiers. Supposons que nous le trouvions égal au produit :

$$2^{1981027125} \times 2^{53} \times 3^{1981027125} \times 2^{11}$$

Cette décomposition signifie que la démonstration Δ comporte deux étapes : l'une correspond au nombre 1981027125×2^{63} et l'autre à 1981027125×2^{11} . Décomposant à nouveau chacun de ces nombres en facteurs premiers, nous obtenons :

$$2^{53} \times 3^5 \times 5^3 \times 7^2 \times 11^3 \text{ et } 2^{11} \times 3^5 \times 5^3 \times 7^2 \times 11^3.$$

Passant maintenant au codage sur l'alphabet de (AF), on trouve que ces deux nombres correspondent à :

$$53, 5, 3, 2, 3, \text{ et à } 11, 5, 3, 2, 3,$$

c'est-à-dire finalement que le nombre de départ correspond à : « du mot $x \times 0 = 0$, on peut déduire le mot $1 \times 0 = 0$ ».

Une signification dans méta-(AF) du nombre initial de (AF) est ainsi obtenue.

a) Le lemme de Gödel.

A tout nombre T , associé à une démonstration, correspond donc naturellement, par ce procédé, le théorème t qu'il permet finalement de démontrer dans (AF). Le système (AF) ainsi « Gödelisé » sera appelé le système (AFG). Puisqu'à chaque axiome et à chaque règle de (AFG) correspond une opération arithmétique, il est possible de savoir, **par examen systématique**, si un nombre T donné correspond effectivement à une démonstration d'un théorème t , T et t forment alors un **couple de nombres conjugués**. L'énoncé « T et t sont conjugués » est également exprimable à l'intérieur même du système (AFG). C'est-à-dire qu'il existe un nombre de Gödel, disons $D(T, t)$, qui représente numériquement cette affirmation.

Nous arrivons maintenant au point crucial de la démonstration de Gödel. Soit A un énoncé du système (AFG) qui comporte une variable libre. A cette variable libre, il est possible de substituer un terme quelconque. En particulier, il est possible de **substituer l'énoncé A lui-même**. Le nombre-énoncé A joue alors deux rôles différents (Cf. Cantor et Richard) c'est à la fois l'énoncé de départ et le terme substitué. Appelons substitution $-*$ notée S^* cette substitution particulière. La formule $S^*(A, n)$ signifiera que n est le nombre de Gödel obtenu quand on effectue, sur l'énoncé A , la substitution $-*$:

$$\text{« } S^*(A, n) \iff n \text{ est le } *-\text{substitué de } A \text{ »}.$$

Gödel construit alors un énoncé (dont on ne sait si c'est un théorème ou un non-théorème) dans lequel il fait intervenir cette originale substitution S^* . Cet énoncé e s'écrit :

$$\neg((\exists T)(\exists t) D(T, t) \wedge S^*(F, n))$$

a) Substitution finale : Dans (AFG) cet énoncé est complètement numérisé et soit E son nombre de Gödel. Comme e comporte la variable libre F nous sommes en droit de prendre la substitution $-*$ sur e en remplaçant F par le nombre E et il vient, en notant t le nombre $*-\text{substitué de } E$:

$$\neg((\exists T)(\exists t) D(T, t) \wedge S^*(E, t))$$

Soit e^* ce second énoncé et E^* son nombre de Gödel. Interprétons e^* . Dans un premier temps e^* signifie que :

- 1) Il n'existe pas de couple (T, t) pour lequel on ait à la fois : T est le nombre de la démonstration arithmétisée du théorème t lui-même arithmétisé et : $S^*(E, t)$ c'est-à-dire : « t est le $* -$ substitué de E^* ». Mais, comme, assurément, S^* est une transformation comme les autres, elle est exprimable en termes de $+$, \times , $/$, ... et de leurs codages en nombres G , un tel nombre t existe. Dès lors c'est le nombre T qui, peut-être, n'existe pas. Une nouvelle interprétation de e^* devient en tous cas :
 - 2) Il n'existe pas de démonstration arithmétisée T pour le théorème t qui est le $* -$ substitué de E . Or, s'il n'y a pas de démonstration c'est bien parce que t lui-même n'est pas un théorème. D'où la troisième interprétation :
 - 3) L'énoncé dont le nombre de Gödel est le $* -$ substitué de E n'est pas un théorème de (AFG). Mais, à ce point, la trappe se referme car, par construction, c'est e^* lui-même qui est le $* -$ substitué de E , en effet le nombre t n'est autre, par construction, que le nombre E^* lui-même.
- Il vient donc la dernière interprétation de e^* :
- 4) « e^* n'est pas un théorème de (AFG). »

Finalement, l'énoncé e^* affirme tout simplement :

« Je ne suis pas un théorème »

Dès lors, de deux choses l'une :

- ou bien e^* est un théorème, mais alors le système (AF) devient **contradictoire**, puisqu'on a simultanément e^* et $\neg e^*$ qui sont démontrables ;
- ou bien e^* est un non-théorème. Dans ce cas, il n'y a pas de contradiction. Mais, en revanche, ce qu'affirme e^* est vrai bien que non démontrable ! Il existe donc alors dans le système une zone de vérités **inatteignables**. On arrive, après avoir prouvé que $\neg e^*$ lui-même n'est pas démontrable, au deuxième théorème de Gödel qui s'énonce ainsi :

Deuxième théorème de Gödel (1931) :
 « Si l'arithmétique formelle est non contradictoire, alors elle n'est pas complète. »

(Il existe au moins un énoncé e^* tel que : $\nvdash e^*$, $\neg \nvdash e^*$).

Comme, dans l'interprétation naturelle de l'arithmétique - dans laquelle au symbole \neg on fait correspondre la négation - on postule précisément : « soit e^* est VRAI, soit e^* est FAUX (c'est-à-dire $\neg e^*$ est VRAI) », c'est-à-dire la non-contradiction. Il vient : **L'arithmétique n'est pas complètement démontrable**. Il existe des vides dans la formalisation et ceci s'étend à tous les systèmes formels qui incluent les axiomes de l'arithmétique. Ainsi, la notion de vérité n'est pas complètement formalisable et la non-contradiction d'une théorie ne peut être démontrée dans cette théorie. La figure 3.7 résume la situation.

Notes :

Le canevas de cette démonstration a été donné car son principe même est à la fois étrange, rigoureux et fondamental. Il fait toucher du doigt les limites cachées, mais internes, des systèmes de manipulation de symboles un tant soit peu puissants, c'est-à-dire ceux qui incluent les nombres. Il ne faut pas être étonné de ne pas suivre cette démonstration en première lecture : reprenez-la à partir de son tableau de principe en 5.3 et dites-vous qu'elle a d'abord été attaquée par de nombreux mathématiciens ; J. Ladrière cite les « réfutations » de Pereiman, Barzin, Kurzynski.

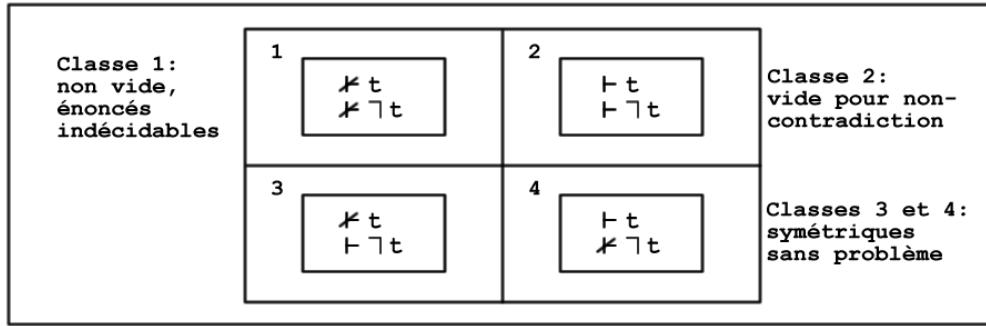


FIG. 3.7 – Second théorème de Gödel : la classe 1 n'est pas vide (Notation : $\vdash t$ signifie t est un théorème, $\nvdash t$ signifie t est un non-théorème).

Notons enfin que, si l'on peut songer à ajouter au système (AF) de départ, l'axiome e^* ou encore l'axiome $\neg e^*$, comme on fait pour obtenir des géométries non euclidiennes, on montre que cela ne sert à rien ici, car on exhibe alors à nouveau, par la même méthode, une proposition e^{**} qui est indécidable.

3.6 Les théorèmes de limitation dans les systèmes formels

Ce théorème de Gödel, qui a été le **premier grand théorème de limitation** obtenu et qui a connu un énorme retentissement à la fois en mathématiques et en philosophie des sciences, n'est pas le seul théorème de limitation. Aussi donnons deux autres résultats, dus à Tarski et à Church, qui sont d'une grande importance théorique et pratique.

Théorème de Tarski (1935) :
 « Il existe des systèmes formels pour lesquels toute interprétation conduit à des énoncés à la fois vrais et non dérivables »

La définition d'une interprétation a été donnée en 4.1.

Ce théorème est le pendant sémantique (c'est-à-dire relatif aux interprétations) du second théorème de Gödel. Succinctement ce résultat affirme : « Ce qui est vrai n'est pas toujours démontrable » ou encore « la notion de vérité n'est pas formalisable ». Le théorème ne dit pas que ce sont toujours les mêmes énoncés qui sont en cause mais que, pour chaque interprétation, il existe au moins un mot, toujours interprété comme VRAI, qui n'est pas un théorème de la théorie.

Les théorèmes de Gödel et Tarski font intervenir l'un et l'autre les interprétations. Pour Tarski, c'est évident et le résultat de Gödel s'appuie sur le sens habituel du symbole \neg (la négation) et le postulat de non-contradiction, (on ne peut pas dériver à la fois t et $\neg t$, sinon on montre facilement que tout mot est un théorème. De tels S.F. n'auraient guère d'intérêt).

Le troisième théorème, celui d'Alonzo Church est, lui, durement syntaxique. C'est, des trois résultats, le plus fondamental.

Théorème de Church (1936) : Le calcul des prédictats du premier ordre est indécidable.
 « Il existe des systèmes formels indécidables ».

Autrement dit, il est impossible de construire, pour certains SF, des procédures capables de séparer les théorèmes des non-théorèmes. On dit aussi que ces SF ne sont pas résolubles et que ces systèmes formels ne sont pas récursivement énumérables. Bien sûr, on voudrait éviter de tels SF, mais, en réalité, on ne sait rien faire sans eux, puisque le calcul des prédictats du premier ordre (PP) lui-même est indécidable, comme le montre Church. A fortiori le système des Principia Mathematica de Russell et Whitehead, plus puissant, est indécidable. Tarski montre de même que la théorie des groupes, celle des anneaux, celle des corps sont non-résolubles. Mais il montre aussi que la géométrie projective réelle et la théorie des corps réels fermés sont résolubles. En outre, on prouve que **(PP) est semi-décidable**, c'est-à-dire que pour les énoncés t qui sont effectivement des théorèmes, on possède une **procédure finie** qui les démontre. Toutes ces preuves reposent sur la définition des procédures de décisions. Church identifie la notion de procédure effective ou calculable à celle **d'algorithme**, c'est-à-dire à celle de fonction récursive (« thèse » de Church).

Indépendamment, dès 1937, Turing formalise l'idée de machine et ramène le problème de la décidabilité à celui de l'arrêt en temps fini d'une machine idéale (Cf. chapitre ???). Les deux approches se révèlent finalement identiques et montrent qu'il n'existe en général : **ni fonction récursive, ni machine de Turing**, pour dire si un mot m est ou n'est pas un théorème d'un système formel donné (d'un « langage » donné). En particulier, le problème de l'arrêt d'un programme sur une machine de Turing est indécidable.

L'ensemble de ces théorèmes de limitation peut être rapproché de ce qui se passe en physique avec le principe d'incertitude d'Heisenberg, curieusement établi quelques années plus tôt (1927) en mécanique quantique : certaines grandeurs physiques sont couplées de telle sorte que, s'il n'existe aucune limite à la précision avec laquelle l'une d'entre elles peut être mesurée, en revanche le produit des précisions est, lui, nécessairement supérieur à une limite (constante de Planck), si bien que, lors de la mesure, augmenter la précision de l'une des grandeurs ne peut que faire baisser la précision sur l'autre. Ainsi, les précisions Δs et Δp , sur les composantes spatiales s et sur les composantes d'impulsion p d'une particule obéissent à la relation d'incertitude inhérente à toute observation :

$$\Delta s \times \Delta p > h.$$

Des philosophes et des informaticiens comme J.R. Lucas, E. Nagel et J.R. Newman ont utilisé les théorèmes de limitations pour faire valoir que ces limitations s'appliquaient directement aux possibilités des machines à traiter les symboles qu'étaient les ordinateurs naissants. Ils en déduisaient, un peu rapidement à mon sens, la **supériorité intellectuelle de l'homme sur la machine**. De tels arguments ont encore cours aujourd'hui, et d'aucuns en profitent pour condamner définitivement toutes les recherches en intelligence artificielle. En réalité, ce qu'affirment les résultats de Gödel, Tarski et Church est très simplement que certains problèmes sont insolubles par les moyens mathématiques actuels. Il s'agit là d'une limite qui vaut tant pour nous que pour les ordinateurs, tant pour le mathématicien que pour les programmes. Et ce n'est pas parce que, aux environs de 1930, ces théorèmes ont été obtenus que toute recherche en mathématique a cessé, bien au contraire ! Quelques problèmes sont impossibles, mais les autres méritent d'être résolus. En outre, savoir quels problèmes exactement sont difficiles, et pourquoi ils le sont, constitue une nouvelle voie de recherche. Les hypo-

thèses de Goldbach, le grand « théorème » de Fermat ne sont toujours pas classés (démontrables ou indécidables ?).

Le dixième problème de Hilbert posé avec 22 autres en 1905, qui s'énonce : « Etablir une procédure effective pour déterminer une solution d'une équation en nombres entiers quelconque » a été démontré **insoluble** par Yuri Matlyasevich en 1970...

La célèbre **conjecture des quatres couleurs** : « Colorier une carte de géographie avec quatre couleurs, de façon que deux pays voisins soient toujours coloriés différemment » (Cf. chapitre 5) a été prouvée, avec l'aide d'un ordinateur, en 1976, par Appel et Haken, pas moins de 120 années après avoir été posée par F. Guthrle.

Si un problème vous semble ardu, assurez-vous d'abord que vous n'êtes pas dans un des cas connus d'indécidabilité, et sinon n'abandonnez pas trop vite !

Il existe des systèmes formels où :

- Gödel : il existe des mots m tels que ni m , ni $\neg m$ ne sont démontrables.
- Tarski : pour toute interprétation on a des énoncés vrais non démontrables.
- Church : il n'existe pas d'algorithme pour séparer les théorèmes des non-théorèmes.

TAB. 3.2 – Les théorèmes de limitation.

3.7 L'algorithme d'unification

L'algorithme qui est décrit dans ce paragraphe répond à la question : « Comment appliquer un théorème à une expression ? ». Cet algorithme est totalement indépendant du système formel dans lequel on se place. Pour un théorème et une expression donnés, il procède de façon rigoureuse par un double parcours dans un ordre préfixé sans choix ni retour arrière. Sa complexité est linéaire en fonction du nombre total de variables substituables dans le théorème et dans l'expression. En fait nous l'avons déjà implicitement utilisé dans les paragraphes précédents pour établir quelques démonstrations dans différents SF.

Les mathématiciens ne semblent pas l'utiliser, même dans les traités de logique. La découverte de cet algorithme en 1966 par Jacques Pitrat dans sa thèse et, indépendamment, par John A. Robinson, a pourtant provoqué une nette discontinuité, avec croissance brusque des performances, dans les programmes de démonstration de théorèmes. C'est un algorithme fondamental en intelligence artificielle.

3.7.1 Le problème

Etant donné un théorème T , règle de réécriture ou production à un antécédent :

$$\begin{array}{ccc} H & \rightarrow & C \\ (\text{hypothèse}) & \text{implique} & (\text{conclusion}) \end{array} \quad (T)$$

et une expression E , il s'agit de voir s'il est possible de rendre H et E complètement identiques, c'est-à-dire *d'unifier* H et E par une **suite de substitutions des variables libres** dans H et E . Si cela est possible, ces substitutions effectuées dans C donnent la nouvelle forme de l'expression E après application du théorème T , par utilisation du modus ponens.

Exemples :

- 1) Soit l'identité d'algèbre bien classique (I1) :

$$(a + b)^2 \equiv a^2 + 2ab + b^2 \quad (\text{I1})$$

considérée comme une abréviation commode de la règle de réécriture : « Si une expression est le carré d'une somme de deux termes, alors on peut la réécrire sous la forme du carré du premier terme, augmenté du double du produit des deux termes, augmenté du carré du second terme ».

Soit, par ailleurs, l'expression (E1) :

$$x^2 + (y + \sqrt{3})^2 \quad (\text{E1})$$

Nous sommes habitués depuis longtemps à faire visuellement coïncider les quantités entre parenthèses dans E1 et I1 : la substitution de y à a et de $\sqrt{3}$ à b permet d'unifier $(y + \sqrt{3})$ et le membre gauche de notre identité.

- 2) De même, la règle de réécriture bien connue (I2) :

$$\sin^2 u + \cos^2 u \mapsto 1 \quad (\text{I2})$$

« s'applique » à l'expression (E2) :

$$\sin^2 3x + \cos^2 3x \quad (\text{E2})$$

pour montrer que cette expression est elle aussi identique à 1. Mais, bien sûr, l'unification échoue pour la même règle et l'expression (E3) :

$$\sin^2 3x + \cos^2 x \quad (\text{E3})$$

Elle échoue aussi avec l'expression (E4) :

$$\cos^2 3x + \sin^2 3x \quad (\text{E4})$$

car il faudrait ici faire jouer auparavant la commutativité du $+$; cela peut se faire formellement grâce à la règle :

$$a + b \longmapsto b + a$$

Une première application de cette règle à (E4) redonne alors (E2) et (I2) peut être appliquée à son tour.

- 3) Ces premiers exemples donnaient lieu à des unifications particulièrement simples. Il n'en est pas toujours ainsi. Apparaît, en général, le problème du choix des substitutions : plusieurs façons différentes peuvent, en effet, exister pour faire coïncider par substitutions deux formules ; les unes donneront des formules qui seront seulement des cas particuliers, c'est-à-dire seront déductibles de la formule la plus générale. Ainsi, soit en logique des propositions l'expression (E5), qui est un théorème d'après le premier axiome de (LP) :

$$(p \supset (q \supset p)) \quad (\text{E5})$$

et soit le métathéorème (T1) :

$$((P \supset Q) \supset R) \longrightarrow (Q \supset R) \quad (\text{T1})$$

Un jeu possible de substitutions non optimales va d'abord être donné, il conduit à une expression-résultat juste, mais inintéressante car trop spécifique, comme nous le verrons. Rappelons la règle du jeu : dans (LP) les lettres propositionnelles $p, q, r\dots$ et les mots $P, Q, R\dots$ sont des variables libres. Toutes les occurrences d'une variable sont remplaçables par un même terme si celui-ci ne contient pas la variable substituée. Pour faire coïncider (T1) et (E5), on peut d'abord tâtonner : ainsi, la substitution qui change Q en q dans $T1$ donne :

$$((P \supset q) \supset R) \longrightarrow (Q \supset R) \quad (\text{E6 } \{Q/q\})$$

Changeons, par ailleurs, dans (E5) p en $P \supset q$ de façon à unifier les premiers symboles des expressions (E5) et (E6) :

$$(P \supset q) \supset (q \supset (P \supset q)) \quad (\text{E7 } \{p/P \supset q\})$$

Il y aura coïncidence entre (E5) substituée et le membre gauche de (T1) substitué, sous réserve de changer encore R en la quantité ($q \supset (P \supset q)$). C'est le nouveau membre droit de (E6) qui, comme dans les exemples précédents, donne le résultat cherché. Obtenu par simple copie, il se trouve être ici :

$$(q \supset (q \supset (P \supset q))) \quad (\text{E8 } \{R/(q \supset (P \supset q))\})$$

Mais, nous allons voir que, par ailleurs, **sans aucun tâtonnement**, l'algorithme d'unification donnera, quant à lui, le résultat :

$$(p \supset (q \supset (P \supset p))) \quad (\text{E9})$$

On voit que ce dernier résultat est plus général. Le précédent étant nécessairement plus faible puisqu'il correspond au cas particulier $p \equiv q$.

Nous allons présenter maintenant cet algorithme optimal d'unification. Son intérêt est double : il donne une méthode systématique et directe de mise en coïncidence par substitutions de deux formules quelconques : les substitutions obtenues sans choix sont les plus générales possibles.

3.7.2 Principe de l'algorithme

L'expression E et le théorème T vont être parcourus en parallèle et unifiés à mesure. Seules les **unifications absolument nécessaires** seront effectuées. Si l'on parvient ainsi à la fin de la partie hypothèse H de T alors l'unification est réussie et le résultat se trouve dans C , modifiée par les substitutions. Sinon il y a échec.

Rappel :

Dans toute expression seules sont **substituables les variables libres** c'est-à-dire non quantifiées par un \forall ou un \exists . Les seules substitutions autorisées sont celles qui concernent le remplacement d'une variable libre x de E ou H par un terme t de E

ou de H . Il n'y a qu'une interdiction, le terme t ne doit pas lui-même contenir déjà la variable x . C'est pourquoi, afin d'éviter toute ambiguïté, le premier travail est de **renommer** les variables de E ou de T de sorte qu'aucun nom ne soit, au départ, commun.

Les substitutions ne seront effectuées que lorsqu'elles seront absolument **indispensables** pour pouvoir progresser dans E et H à la fois. Ces substitutions seront les seules effectuées. Les symboles représentant autre chose que des variables sont non substituables : il s'agit soit d'opérateurs (comme $+$, \times , \supset , $\sin\dots$) soit de constantes (comme 1, 2, $\Pi\dots$). De tels symboles doivent alors être rencontrés en parallèle aux mêmes places dans E et H , sinon aucune unification n'est possible.

L'idée fondamentale de l'algorithme est liée à une procédure de parcours convenable : opérateur principal d'abord, puis chacun des sous-termes qu'il commande, le plus à gauche en premier par exemple. Chaque sous-terme est alors parcouru pareillement : opérateur principal, sous terme et ainsi de suite jusqu'à atteindre une variable ou une constante. La représentation des expressions en arbres est donc bien adaptée (Voir figure 3.8).

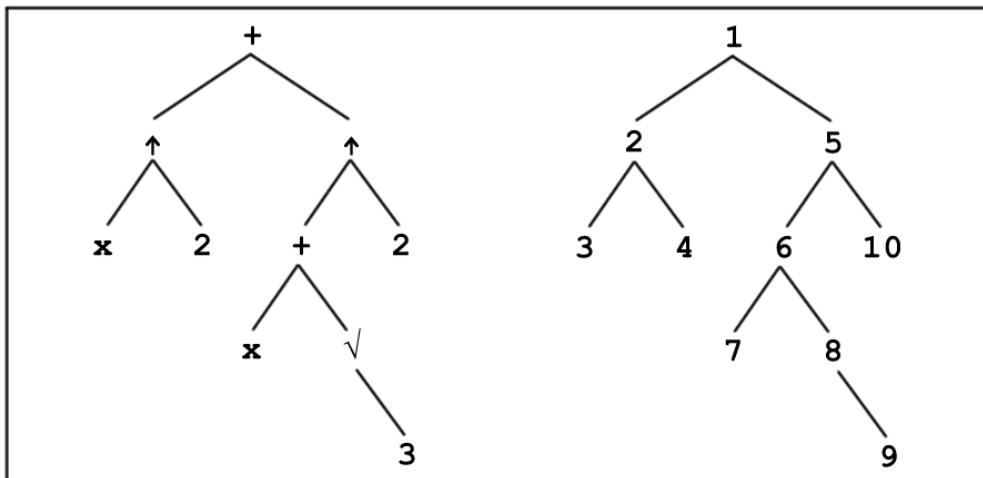


FIG. 3.8 – Représentation et ordre de parcours de l'expression : $x^2 + (x + \sqrt{3})^2$ pour l'algorithme d'unification ; ordre antérieur ou préfixé : chaque symbole est traité dès qu'il est rencontré par descente de haut vers le bas et de la gauche vers la droite, (dans l'ordre des numéros de la figure de droite).

Moyennant cet ordre de parcours, « profondeur d'abord », introduisons pour la suite, les indices e et h :

e = indice du symbole courant dans l'expression E ;
 h = indice du symbole courant dans l'expression H .

La situation générale, l'hypothèse de récurrence de l'algorithme, sera dès lors : **tous les symboles de E et de H coïncident, jusqu'à e et h non compris : on tente de faire coïncider les symboles d'indices e et h dans E et H respectivement**. Il y a quatre cas possibles suivant que e et h représentent chacun une variable substituable ou non. Si une substitution doit être faite, elle sera effectuée par une procédure spécifique nommée « SUB ». Pour commencer, écrivons le texte de cette procédure SUB.

3.7.3 La procédure de substitution

Les données de SUB seront : le nom v de la variable qui doit être remplacée, le terme t qui la remplace, les deux indices e et h . Après substitution, Il y a coïncidence de E et T jusqu'aux termes qui commencent en e et h compris. Le texte de cette procédure sera ainsi celui de l'encadré suivant. – Les noms soulignés correspondent à des fonctions élémentaires non détaillées ici, car elles dépendent du choix d'implémentation. – Les indices e et h progressent lors d'un appel à la procédure SUB.

Procédure SUB :

DONNEES : v, t, e, h : variable libre, terme,
indice dans E , indice dans H .

RESULTATS : FAUX si la substitution est impossible
VRAI sinon, et nouveau E , nouveau Γ ,
nouveau o , nouveau h .

SUB \leftarrow VRAI

PARCOURIR t à partir du début

SI symbole-courant = v ALORS SUB \leftarrow FAUX

| RETOUR (cas d'échec 1)

FSI

FIN P

$e \leftarrow$ indice – terme – suivant – terme – commençant en e

$h \leftarrow$ indice – terme – suivant – terme – commençant en h

{Les parties déjà traitées en E et H ne sont plus
d'aucune utilité : la mise à jour en e et h n'est
donc pas nécessaire. On sait que les débuts de E
et H coïncident.}

PARCOURIR E à partir de e

SI symbole-courant = v ALORS remplacer \leftarrow FAUX

| symbole-courant par t

FSI

FIN P

PARCOURIR T à partir de h

SI symbole-courant = v ALORS remplacer \leftarrow FAUX

| symbole-courant par t

FSI

FIN P

Mémoriser la substitution (v, t)

FSUB

Exemple de substitution :

Si, dans l'expression précédente (figure 3.8), on désire substituer $(y + \sqrt{3})$ à x , la procédure SUB donne avec :

$$v = x, t = y + \sqrt{3}, e = 3, h = 1$$

PARCOURS de t : substitution possible ;
 $e \leftarrow 4$ indice du terme suivant dans E ;
 $h \leftarrow 5$ indice du terme suivant dans H ;



FIG. 3.9 – Représentation de $t \equiv y + \sqrt{3}$

PARCOURS de E à partir de e : le symbole courant x est rencontré à nouveau à l'indice 7 on le remplace par t . L'expression devient celle de la figure 3.10.

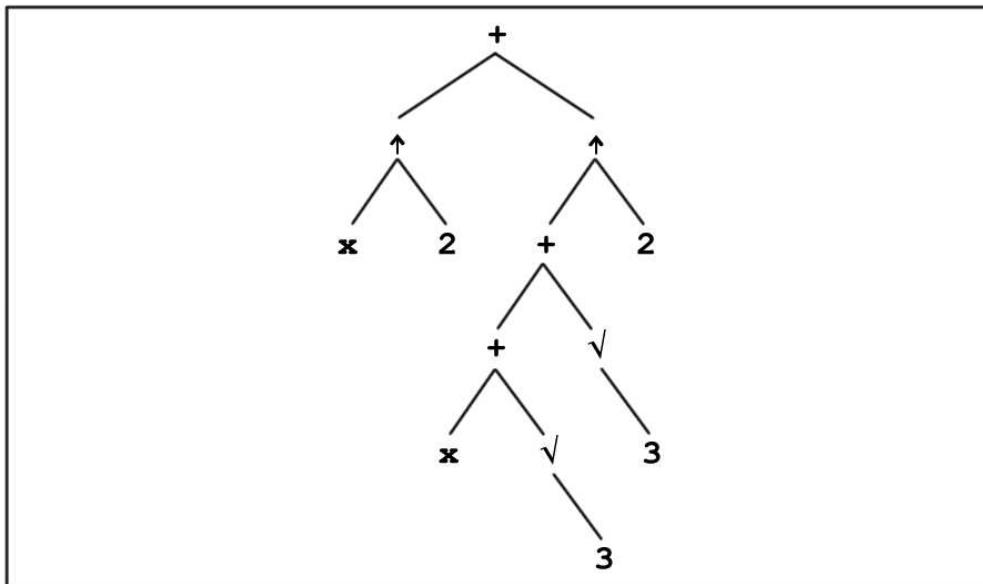


FIG. 3.10 – Nouvelle expression E après substitution.

PARCOURS de T : sans objet pour l'instant.

Fin de la procédure SUB : x n'a été remplacée que dans la partie utile des expressions, les débuts sont systématiquement supposés déjà unifiés, par construction de l'algorithme d'unification lui-même, que nous décrivons maintenant.

Procédure UNIFICATION

DONNEES : E : expression, $T = (H, C)$ théorème = (hypothèse, conclusion)

RESULTAT : VRAI et nouveau $E \equiv$ nouveau C , avec la liste des substitutions effectuées si unification réussie.

FAUX sinon

$e \leftarrow$ **début-expression** (E) ($s(e) =$ symbole en e dans E)

$h \leftarrow$ **début-expression** (H) ($s(h) =$ symbole en h dans H)

PARCOURIR parallèlement E et H , à partir de e et h ,

TANT QUE $h \leq$ fin-expression (H)

SI $s(e) \neq s(h)$ ALORS

{S'il y a déjà égalité, on continue le parcours directement}

{D'abord les deux cas où le symbole en e est un opérateur :
les constantes sont ici considérées comme des opérateurs}

SI $s(e) =$ opérateur ALORS

SI $s(h) =$ opérateur ALORS unification \leftarrow FAUX

| RETOUR {cas échec 2}

SINON $t \leftarrow$ **terme-débutant – en e dans E** ;

| $\{s(h)$ est une variable libre à laquelle on
substitue le terme $t\}$

SUB ($s(h), t, e, h$).

FSI

SINON {cas où le symbole en e est une variable

SI $s(h) \neq$ opérateur ALORS **SUB** ($s(h), s(e), e, h$)

| {variable contre variable : simple
changement de l'un des noms}

SINON $t \leftarrow$ **terme-debutant en h dans H** ;

| **SUB** ($s(h), t, e, h$).

FSI

FSI

SI SUB = FAUX ALORS UNIFICATION \leftarrow FAUX ; RETOUR ;
FSI

FSI

FIN

UNIFICATION \leftarrow VRAI

RETOUR : liste des substitutions effectuées.

{La nouvelle expression E est la partie conclusion C
de T dans laquelle les substitutions ont été faites}.

FIN UNIFICATION.

3.7.4 Finitude de l'algorithme d'unification

L'arrêt a lieu lorsque l'on rencontre la fin de H , E pouvant être plus longue que H si T est une règle de réécriture. Cependant comme les sous-arbres droits de E et H peuvent croître tous les deux. Il reste à montrer que le processus est bien fini.

Le nombre de variables substituables dans E et H est, bien entendu, fini au départ et :

- d'une part, on ne crée pas de nouvelle variable en cours d'algorithme, d'autre part :
- chaque substitution — qui remplace une variable par un terme ne la contenant pas
 - a pour effet de diminuer d'une unité le nombre total de variables à traiter. L'algorithme converge donc avec un nombre de substitutions au plus égal au nombre

initial de variables libres dans E et H .

Exemples d'exécution :

- | | | |
|---|---|--|
| 1) $E \equiv a + b$ | $\left. \begin{array}{l} \\ \end{array} \right\}$ | Cas d'échec 2 :
opérateurs non unifiables |
| 2) $E \equiv \sin^2 a + \cos^3 a$ | | idem (opérateurs = constantes) |
| 3) $E \equiv \log \sqrt{a} - \log \sqrt{2 * a}$ | $\left. \begin{array}{l} \\ \end{array} \right\}$ | Cas d'échec 1 (dans SUB)
$T \equiv x - x \rightarrow 0$ |

* On substitue d'abord $\log \sqrt{a}$ à x , T devenant : $\log \sqrt{a} - \log \sqrt{a} \rightarrow 0$; les log et les $\sqrt{}$ s'unifient, mais il faudrait alors remplacer les a par des $2 * a$: cela est interdit (échec 1 dans la procédure SUB).

4) Reprenons l'exercice de logique des propositions, déjà étudié en début de paragraphe, dans lequel on avait :

$$E \equiv (p \supset (q \supset p)),$$

et $T \equiv ((P \supset Q) \supset R) \rightarrow (Q \supset R)$

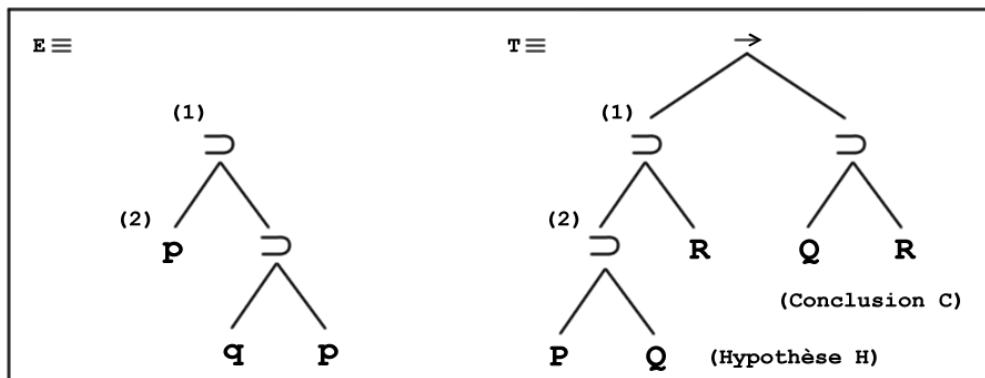


FIG. 3.11 – Unification en logique des propositions.

Nous avons utilisé la représentation arborescente des expressions pour suivre des yeux le déroulement de l'algorithme (Figure 3.11)

Les deux premiers symboles dans E et H sont le même opérateur \supset : on progresse dans les deux arbres. Dans E on rencontre une variable – soit p – alors que dans H la connective 3 indique le début d'un terme t , il faut donc changer p en t où t est le sous-arbre :

Dans E , on rencontre donc maintenant une connective, contre une variable, R , dans T .

On a donc une unification réussie ; en revenant à la notation usuelle, l'expression E a été changée en F :

$$(Q \supset (q \supset (P \supset Q))) \text{ par les substitutions } p/P \supset Q \text{ et } R/q \supset (P \supset Q)$$

Autrement dit, l'expression E et la partie hypothèse H de T ont été unifiées en :

$$(P \supset Q) \supset (P \supset (P \supset Q))$$

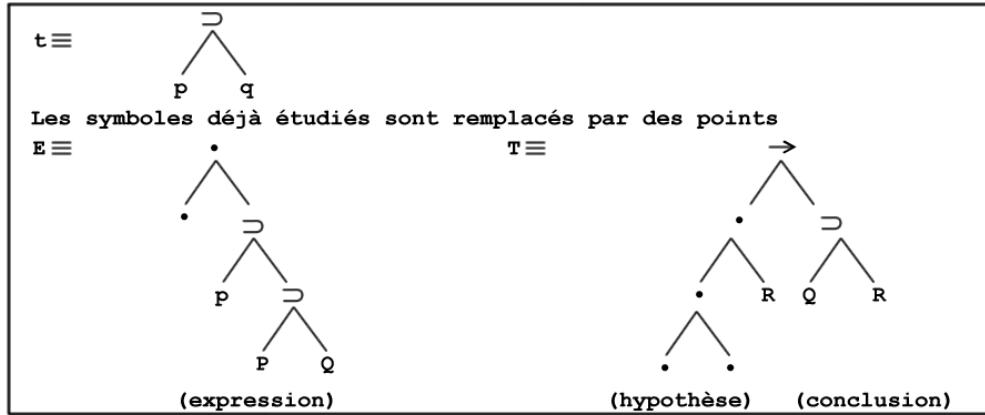


FIG. 3.12 – Unification (suite)

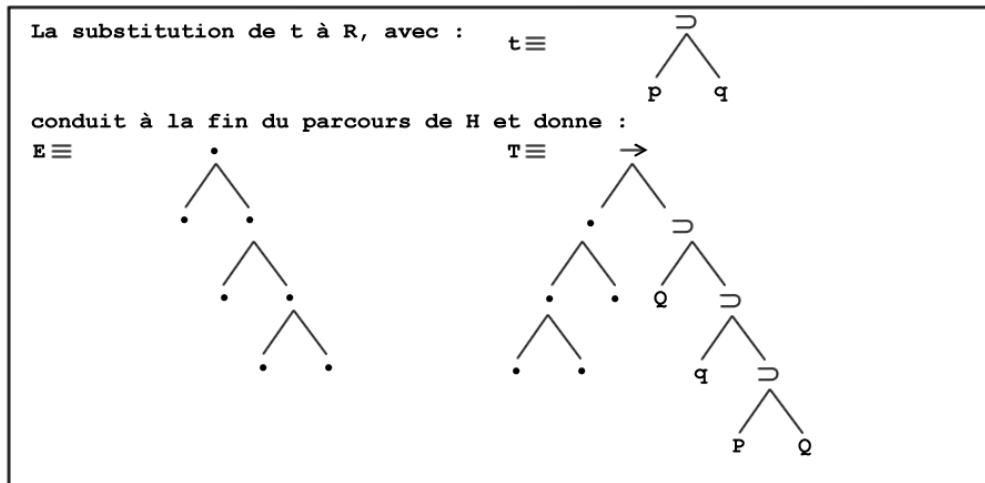


FIG. 3.13 – Unification (fin)

pour donner l'expression F . En normalisant, on est donc parvenu à partir de E et de T , au résultat :

$$(p \supset q) \supset (r \supset (p \supset q)) \rightarrow (q \supset (r \supset (p \supset q)))$$

Le membre gauche de cette production est le théorème le plus général que l'on peut obtenir à partir de E et T . Notamment le résultat, issu de tâtonnements au paragraphe 7.1. n'en est qu'un cas particulier.

3.7.5 « Variables » non substituables : indéterminées.

Nous avons, jusqu'ici, supposé que les variables dans l'expression E et dans le théorème T jouaient le même rôle. Cela n'est pas toujours vrai. En effet, quel que soit le système formel où l'on travaille, les variables de T sont, par définition, dans tous les cas, substituables. Cela est dû au fait que ces variables, à l'origine, sont quantifiées universellement et deviennent libres par l'axiome de particularisation. En revanche, il

est courant de rencontrer des contextes dans lesquels les « variables » de E **ne sont pas substituables**.

Le cas où $s(e)$ est une variable de E et où H marque le début d'un terme correspond alors à un troisième cas, où l'unification est impossible (dernière substitution interdite dans la procédure : cas d'échec 3). Cela tient au fait que les variables de E , contrairement à celles intervenant dans un théorème, ne sont **pas toujours libres**, même si souvent **les quantificateurs sont omis par souci de concision**. Une expression est en général manipulée pour un calcul, une évaluation (Dans l'expression $3x^2+4x-6 = 0$ par exemple, x n'est certes pas une variable et toute substitution sur x est illégale). Les variables qui y figurent sont donc, à mesure, de plus en plus contraintes et on ne peut légalement plus leur substituer n'importe quel terme. En particulier, il arrive que les différentes « variables » deviennent interdépendantes. Les variables de tout **théorème** sont des variables **libres** (ou muettes). Celles d'une **expression** sont, en général, des variables **liées**. On parle dans ce cas de *demi-unification ou filtrage*. Ainsi, en algèbre, la règle de simplification :

$$T \equiv a + (b - a) \mapsto b$$

est légale. Quand on cherche à l'appliquer à une expression courante comme :

$$E \equiv x + y$$

une première substitution, où a est changé en x , est valide. T devient $T1$:

$$T1 \equiv x + (b - x) \mapsto b \tag{3.3}$$

une seconde substitution, où y serait changé en $(b - x)$, est ici en principe **interdite**. car le contexte, par exemple la résolution d'un système en x et y , lie les « variables » x et y entre elles. (Si l'on veut cependant forcer cette substitution, il conviendra de retenir la transformation :

$$(y \text{ changé en } b - x)$$

comme une **condition** liant les variables x et y , pour toute la suite de la résolution du problème en cours. Moyennant cette **condition**, E pourra être simplifié en b , b restant une variable libre).

Prenons un second exemple, plus concret. Dans le corps des réels, la règle de réécriture suivante est légale :

$$e^z * e^z \mapsto e^{2z}$$

Elle peut être utilisée pour simplifier l'expression :

$$e^{x-3} * e^{2y-3} \mapsto e^2 \tag{R}$$

Une première substitution change z en $x - 3$ dans T ; T devient :

$$e^{x-3} * e^{x-3} \mapsto e^{2(x-3)} \tag{T}$$

Une deuxième substitution change x en $2y$ dans T : l'unification est réussie et la conclusion de T devient :

$$e^{2(2y-3)}$$

Mais, nous avons implicitement ajouté la condition : $x = 2y$, lors de la dernière substitution.

Par rapport, à la **résolution de l'équation** (R), nous obtenons maintenant :

$$e^{2(2y-3)} = e^2 \text{ soit } y = 2 \text{ (et donc } x = 4)$$

Mais en réalité, (R) peut être résolu directement sans utiliser (T) ; il vient, en effet :

$$e^{x+2y-6} = e^2 \text{ soit } x + 2y = 8$$

Et il est clair que nous venons d'obtenir la solution générale de (R), dont la précédente, engendrée par unification, n'était qu'un cas particulier.

Les symboles x et y , liés entre eux par l'équation (R), ne sont pas des variables, substituables au sens habituel. Si l'on veut cependant leur substituer des termes, il convient de retenir les conditions correspondantes et de discuter a posteriori les solutions obtenues.

Il convient donc de se méfier de ces fausses variables libres qui apparaissent de façon habituelle dans les expressions. Pour les désigner, le mot précis en mathématique existe : ce sont des *indéterminées*. Elles ne peuvent être substituées qu'en testant a posteriori la validité des substitutions. Sans cette précaution, on construit sans peine de faux paradoxes.

3.7.6 Mise en oeuvre de l'unification

L'algorithme d'unification ainsi précisé peut être utilisé de façon systématique dans différents contextes mathématiques : algèbre, arithmétique, trigonométrie, logique, dérivation et intégration formelles. Il s'étend sans difficulté aux règles de dérivation à plusieurs antécédents H_1 et $H_2 \dots$ et $H_n \rightarrow C$.

A chaque fois, cet algorithme sera lancé : pour chacun des théorèmes licites dans le domaine : pour un théorème donné, pour chaque sous-expression de l'expression sur laquelle on travaille. La procédure générale de démonstration est donc : (voir figure 3.14).

La procédure s'arrête quand aucun théorème ne s'applique à aucune sous-expression. L'encadré ci-dessus n'est, bien entendu, qu'un schéma de principe. Il peut être complètement inefficace si l'ensemble τ des théorèmes est important. Certaines règles peuvent être logiquement prioritaires sur d'autres. De plus l'**ordre d'application des théorèmes n'est pas indifférent** : le résultat final E modifié, dépend en général de cet ordre. Si l'on veut manipuler E pour atteindre un but donné, plusieurs essais tenant compte de plusieurs ordres différents dans τ et/ou plusieurs places E différentes dans E devront être envisagées. Il convient donc de piloter intelligemment en fonction du but et du contexte cette procédure générale d'unification.

En outre, un problème tout à fait fondamental se pose : la procédure générale qui précède peut très bien ne pas se terminer. Il en sera ainsi, en particulier, chaque fois que τ contiendra deux règles telles que :

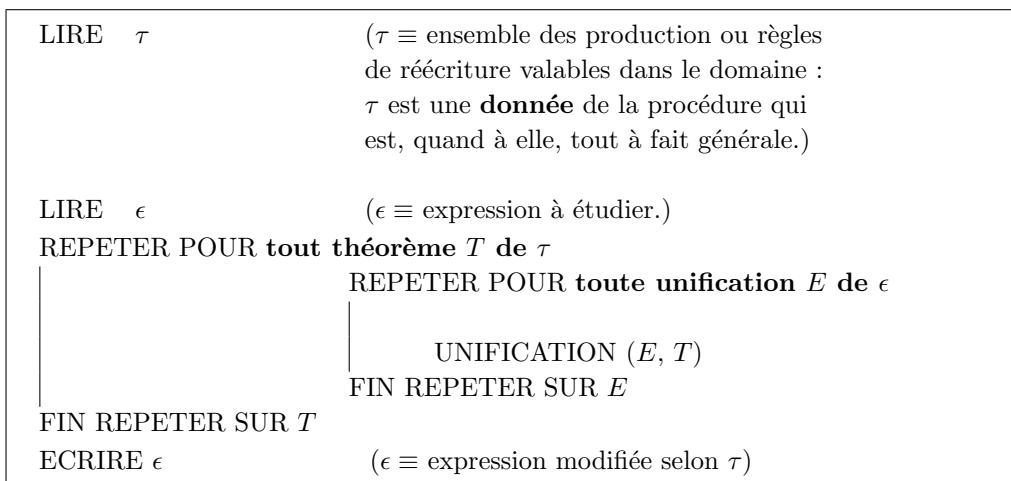


FIG. 3.14 – Procédure générale de démonstration.

$$x - x \mapsto 0 \text{ et } 0 \mapsto x - x \quad (3.4)$$

En fait, ce même phénomène de cyclage peut se produire avec une seule règle, comme :

$$x + y \mapsto y + x \quad (3.5)$$

ou encore :

$$x * y \mapsto y * x \quad (3.6)$$

qui provoquent des permutations sans fin entre deux termes successifs de s , avec un ensemble de règles en nombre quelconque comme :

$$\begin{aligned} u * (v/w) &\mapsto (u * v)/w \\ \tan x &\mapsto \sin x / \cos x \\ (u * v)/u &\mapsto v \\ \sin x &\mapsto \cos x * \tan x \end{aligned}$$

où par utilisation des 4 règles, on boucle sur l'expression $\sin x$.

Ce problème de cyclage peut être résolu en s'inspirant de ce que nous faisons couramment à la main. La solution comporte deux volets : une procédure de *normalisation* de l'expression ; une procédure d'*inhibition* des théorèmes.

Normalisation : Certaines règles ne sont appliquées que sous une condition déterminée. Cette condition interdit à la règle d'être ensuite redéclenchée, ce qui aurait l'effet néfaste de détruire le premier travail. Ainsi, le théorème de commutativité de l'opérateur de multiplication : $y * x \mapsto x * y$ ne sera lancé que si : masse $(x) <$ masse (y) .

Dans cette expression, la masse est une fonction numérique qui traduit l'ordre de standardisation désiré sur les variables qui interviennent dans l'expression. À la main, nous écrivons communément : $2 * a$ et non pas : $a * 2$, ou encore : $x^4 y^2 z$ et non pas : $z^4 x^2 y$.

Inhibition : L'application réussie de certaines règles provoque non seulement la modification convenable par unification, mais aussi interdit, pour un temps, l'application

d'autres règles liées à celle qu'on vient d'unifier avec l'expression courante. Ainsi, les règles : $x - x \mapsto 0$ et $0 \mapsto x - x$ doivent mutuellement s'inhiber.

La règle : $\tan x \mapsto \sin x / \cos x$ doit inhiber de la même façon les deux règles de passage inverse du sinus et du cosinus en termes de tangente :

$$\sin x \mapsto \tan x * \cos x \text{ et } \cos x \mapsto \sin x / \tan x$$

Réiproquement, il est possible que l'application de certaines règles réactive d'autres règles précédemment inhibées. Cela sera le cas lorsque l'on jugera que l'effet d'une règle change « suffisamment » la structure d'une expression pour que tout risque de retour à un état antérieur soit écarté.

Remarque :

Une façon différente, mais souvent plus coûteuse, de procéder pour éviter ces cyclages, est d'écarter a posteriori toute nouvelle expression qui se trouve être identique à une des expressions déjà obtenues.

Moyennant ces deux procédures de « normalisation » et « d'inhibition-activation », l'algorithme d'unification fournit un moyen automatique de manipulation formelle d'expressions. La table qui suit donne par exemple un ensemble de règles de réécriture qui permet d'effectuer en machine toutes les manipulations habituelles du type « simplification » sur une expression algébrique quelconque.

$-0 \mapsto 0$	T_0
$0 + x \mapsto x$	T_1
$0 * x \mapsto 0$	T_2
$1 * x \mapsto x$	T_3
$-(-x) \mapsto x$	T_4
$x * (-y) \mapsto (-x) * y$	T_5
$-(x * y) \mapsto (-x) * y$	T_6
$x * (y * (-z)) \mapsto (-x) * (y * z)$	T_7
$z * (x + y) \mapsto z * x + z * y$	T_8
$x + (-x) \mapsto 0$	T_9
$x + x \mapsto 2 * x$	T_{10}
$C_1 * x + C_2 * x \mapsto (C_1 + C_2) * x$	T_{11}
$x * (y/x) \mapsto y$	T_{12}
$x * (y/z) \mapsto (x * y)/z$	T_{13}
$(x/z) + (y/z) \mapsto (x + y)/z$	T_{14}
$x/1 \mapsto x$	T_{15}
$x/x \mapsto 1$	T_{16}

FIG. 3.15 – Table de règles de réécriture en algèbre en vue de la simplification

(C_1 et C_2 désignent des constantes quelconques, x , y , et z désignent des variables libres ; on suppose que la procédure de réarrangement à l'aide des masses des termes fonctionne : les constantes sont placées devant les variables, $-x$ après $+x$, x avant y etc.).

3.8 Utilisation de l'unification

Je ne suis pas sûr, à ce point de l'exposé, d'avoir réussi à vous faire sentir toute la richesse de cet algorithme d'unification. Vous êtes en train de vous dire : « C'est bien joli, c'est propre et rigoureux, mais c'est encore des mathématiques et ça ne s'applique qu'à des systèmes formels, bien définis ». Eh bien justement non : changeons donc radicalement d'exemples et ouvrez vos yeux !

3.8.1 Problèmes d'analogie.

Vous connaissez ces tests d'intelligence auxquels sont maintenant soumis tous les demandeurs d'emploi ; une famille de figures géométriques est par exemple présentée, telle que celles de la figure 3.17.

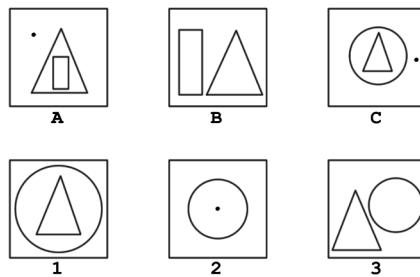


FIG. 3.16 – Test géométriques

On demande de trouver parmi les figures 1, 2, 3 de la rangée du bas, la figure x , telle que « la figure C soit à la figure x ce que la figure A est à la figure B ». Quelle est l'analogie entre A et B d'une part, et C et x d'autre part ? En d'autres termes, il s'agit de voir si l'on peut **unifier** la figure C avec la partie gauche du « théorème » : $A \rightarrow B$. On vérifiera ensuite que le résultat de l'unification est bien une des trois réponses proposées.

Mais, que signifie : « $A \rightarrow B$ » ? A peut être décrit, simplement comme un assemblage de formes, reliées par des relations (exactement comme une formule d'algèbre est un assemblage de variables reliées par des opérateurs). Ainsi :

$A \equiv (\text{point}, \text{triangle}, \text{rectangle})$	ET	(rectangle	INCLUS-DANS	triangle)
	ET	(point	EXTERIEUR	triangle)
	ET	(point	EXTERIEUR	rectangle)

Les mots en majuscules désignent ici les relations dont les significations sont évidentes. Les mots en minuscules correspondent aux variables. Alors, de la même façon et avec la même terminologie, il vient pour B :

$B \equiv (\text{rectangle}, \text{triangle})$	ET	(rectangle	A-GAUCHE	triangle)
	ET	(rectangle	EXTERIEUR	triangle)

En outre, en prenant pour échelles de références celles de la figure A, il convient d'ajouter dans la description de B :

ET (ECHELLE rectangle = 2)

ET (ECHELLE triangle = 1)

Le théorème T n'est maintenant rien d'autre que :

(description de A) \rightarrow (description de B), soit :

A \equiv (point, triangle, rectangle)

ET (rectangle INCLUS DANS triangle)

ET (point EXTERIEUR triangle)

ET (point EXTERIEUR rectangle)

B \equiv (point, triangle, rectangle)

ET (rectangle A-GAUCHE triangle)

ET (rectangle EXTERIEUR triangle)

ET (ECHELLE rectangle=2)

ET (ECHELLE triangle=1)

La description de C est de manière semblable :

C \equiv (cercle, triangle, point)

ET (point EXTERIEUR triangle)

ET (triangle INCLUS-DANS cercle)

ET (point EXTERIEUR cercle)

La question devient :

« Peut-on unifier la description de A avec la description de C ? »

Déroulons notre algorithme d'unification à la main. Remarquons bien, ici, que **l'ordre des sous-expressions n'intervient pas** : une relation de A peut s'unifier avec une relation quelconque de C sans qu'il y ait lieu de se préoccuper de l'ordre dans lequel elles apparaissent dans les deux descriptions. De plus, changeons comme à l'habitude les noms des **variables** de A et B, pour pouvoir faire les substitutions les plus générales possibles dans T . Actuellement, certaines variables dans A et C portent, a priori par hasard, les mêmes noms : il n'y a en effet aucune raison pour que les « triangles » de A et de B soient obligatoirement associés au « triangle » de C. La description de $T \equiv (A \rightarrow B)$ devient :

$$\left\{ \begin{array}{l} A \equiv (a, b, c) \\ \text{ET } (c \text{ INCLUS-DANS } b) \\ \text{ET } (a \text{ EXTERIEUR } b) \\ \text{ET } (a \text{ EXTERIEUR } c) \end{array} \right. \longrightarrow \left\{ \begin{array}{l} B \equiv (c, b) \\ \text{ET } (c \text{ A-GAUCHE } b) \\ \text{ET } (c \text{ EXTERIEUR } b) \\ \text{ET } (\text{ECHELLE } c = 2) \\ \text{ET } (\text{ECHELLE } b = 1) \end{array} \right.$$

Notons que l'hypothèse faite avec ces nouvelles descriptions revient à dire que les formes des objets de A et de B ne comptent pas, seules interviennent leurs relations mutuelles.

3.9 Utilisation de l'unification

Je ne suis pas sûr, à ce point de l'exposé, d'avoir réussi à vous faire sentir toute la

richesse de cet algorithme d'unification. Vous êtes en train de vous dire : « C'est bien joli, c'est propre et rigoureux, mais c'est encore des mathématiques et ça ne s'applique qu'à des systèmes formels, bien définis ». Eh bien justement non : changeons donc radicalement d'exemples et ouvrez vos yeux !

3.9.1 Problèmes d'analogie.

Vous connaissez ces tests d'intelligence auxquels sont maintenant soumis tous les demandeurs d'emploi ; une famille de figures géométriques est par exemple présentée, telle que celles de la figure 3.17.

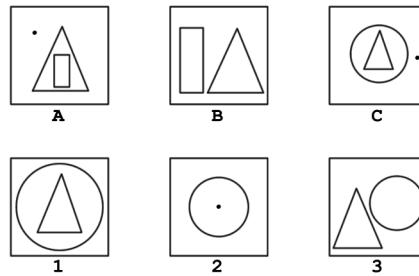


FIG. 3.17 – Test géométriques

On demande de trouver parmi les figures 1, 2, 3 de la rangée du bas, la figure x , telle que « la figure C soit à la figure x ce que la figure A est à la figure B ». Quelle est l'analogie entre A et B d'une part, et C et x d'autre part ? En d'autres termes, il s'agit de voir si l'on peut **unifier** la figure C avec la partie gauche du « théorème » : A \rightarrow B. On vérifiera ensuite que le résultat de l'unification est bien une des trois réponses proposées.

Mais, que signifie : « A \rightarrow B » ? A peut être décrit, simplement comme un assemblage de formes, reliées par des relations (exactement comme une formule d'algèbre est un assemblage de variables reliées par des opérateurs) . Ainsi :

A \equiv (point, triangle, rectangle)	ET	(rectangle	INCLUS-DANS	triangle)
	ET	(point	EXTERIEUR	triangle)
	ET	(point	EXTERIEUR	rectangle)

Les mots en majuscules désignent ici les relations dont les significations sont évidentes. Les mots en minuscules correspondent aux variables. Alors, de la même façon et avec la même terminologie, il vient pour B :

B \equiv (rectangle, triangle)	ET	(rectangle	A-GAUCHE	triangle)
	ET	(rectangle	EXTERIEUR	triangle)

En outre, en prenant pour échelles de références celles de la figure A, il convient d'ajouter dans la description de B :

ET (ECHELLE rectangle = 2)
ET (ECHELLE triangle = 1)

Le théorème T n'est maintenant rien d'autre que :

(description de A) \rightarrow (description de B), soit :

A \equiv (point, triangle, rectangle)

ET (rectangle INCLUS DANS triangle)
 ET (point EXTERIEUR triangle)
 ET (point EXTERIEUR rectangle)

B \equiv (point, triangle, rectangle)

ET (rectangle A-GAUCHE triangle)
 ET (rectangle EXTERIEUR triangle)
 ET (ECHELLE rectangle=2)
 ET (ECHELLE triangle=1)

La description de C est de manière semblable :

C \equiv (cercle, triangle, point)

ET (point EXTERIEUR triangle)
 ET (triangle INCLUS-DANS cercle)
 ET (point EXTERIEUR cercle)

La question devient :

« Peut-on unifier la description de A avec la description de C ? »

Déroulons notre algorithme d'unification à la main. Remarquons bien, ici, que **l'ordre des sous-expressions n'intervient pas** : une relation de A peut s'unifier avec une relation quelconque de C sans qu'il y ait lieu de se préoccuper de l'ordre dans lequel elles apparaissent dans les deux descriptions. De plus, changeons comme à l'habitude les noms des **variables** de A et B, pour pouvoir faire les substitutions les plus générales possibles dans T. Actuellement, certaines variables dans A et C portent, a priori par hasard, les mêmes noms : il n'y a en effet aucune raison pour que les « triangles » de A et de B soient obligatoirement associés au « triangle » de C. La description de $T \equiv (A \rightarrow B)$ devient :

$$\left\{ \begin{array}{l} A \equiv (a, b, c) \\ \text{ET } (c \text{ INCLUS-DANS } b) \\ \text{ET } (a \text{ EXTERIEUR } b) \\ \text{ET } (a \text{ EXTERIEUR } c) \end{array} \right. \longrightarrow \left\{ \begin{array}{l} B \equiv (c, b) \\ \text{ET } (c \text{ A-GAUCHE } b) \\ \text{ET } (c \text{ EXTERIEUR } b) \\ \text{ET } (\text{ECHELLE } c = 2) \\ \text{ET } (\text{ECHELLE } b = 1) \end{array} \right.$$

Notons que l'hypothèse faite avec ces nouvelles descriptions revient à dire que les formes des objets de A et de B ne comptent pas, seules interviennent leurs relations mutuelles.

Commençons à unifier les descriptions de A et de C, suivant l'ordre où elles sont dans C. La relation : « point EXTERIEUR triangle » peut être rendue identique à : « a EXTERIEUR b », en changeant dans A : a en point et b en triangle. Nous savons, à ce stade, qu'il existe une autre possibilité, liée à la seconde occurrence de la relation EXTERIEUR dans A et au fait que l'ordre n'intervient pas : c'est l'unification de « point EXTERIEUR triangle » dans C avec « a EXTERIEUR c » dans A. Laissons pour l'instant en réserve cette seconde possibilité, et considérons donc le premier cas. Les substitutions annoncées donnent comme nouvelle description de A :

$\left\{ \begin{array}{l} A \equiv (\text{point}, \text{triangle}, c) \\ \\ \text{ET } (c \quad \text{INCLUS-DANS} \quad \text{triangle}) \\ \text{ET } (\text{point} \quad \text{EXTERIEUR} \quad \text{triangle}) \\ \quad (\text{unifiée avec la première relation de C}) \\ \text{ET } (\text{point} \quad \text{EXTERIEUR} \quad c) \end{array} \right.$

La relation suivante dans C est : « triangle INCLUS-DANS cercle ». Il n'y a ici qu'une unification possible : elle concerne la relation « c INCLUS-DANS triangle » dans la description de A. Mais nous rencontrons le **troisième cas d'échec** de l'algorithme : il faudrait, pour identifier ces deux expressions, changer, d'une part, c en triangle ; ceci est possible ; mais aussi changer triangle en cercle, ce qui est interdit, car comme déjà signalé (IV.3), dans bien des contextes, les variables de l'expression ici C, ne peuvent pas être considérées comme indépendantes : le Cercle et le Triangle de C ne sont aucunement unifiables, car ils sont interdépendants dans C et jouent en fait le rôle de **constantes**. Il reste la seconde possibilité, mise en attente auparavant :

unifier « point EXTERIEUR triangle » dans C avec : « a EXTERIEUR c » dans A
--

On substitue pour cela : point à a et triangle à c . Le théorème $T \equiv A \rightarrow B$ devient cette fois :

$\left\{ \begin{array}{l} A \equiv (\text{point}, b, \text{triangle}) \\ \text{ET } (\text{triangle} \quad \text{INCLUS-DANS} \quad b) \\ \text{ET } (\text{point} \quad \text{EXTERIEUR} \quad b) \\ \text{ET } (\text{point} \quad \text{EXTERIEUR} \quad \text{triangle})^* \end{array} \right.$
$\longrightarrow \left\{ \begin{array}{l} B \equiv (\text{triangle}, b) \\ \text{ET } (\text{triangle A-GAUCHE} \quad b) \\ \text{ET } (\text{triangle EXTERIEUR} \quad b) \\ \text{ET } (\text{ECHELLE triangle} \quad = 2) \\ \text{ET } (\text{ECHELLE } b \quad = 1) \end{array} \right.$

L'étoile marque une relation déjà unifiée.

La relation suivante dans C est : « Triangle INCLUS-DANS Cercle ». Elle s'identifie de façon **unique** avec la sous-expression « triangle INCLUS-DANS b », en substituant maintenant cercle à b . Cette substitution donne dans T :

$\left\{ \begin{array}{l} A \equiv (\text{point}, \text{ cercle}, \text{triangle}) \\ \text{ET } (\text{triangle} \quad \text{INCLUS-DANS} \quad \text{ cercle})^* \\ \text{ET } (\text{point} \quad \text{EXTERIEUR} \quad \text{ cercle}) \\ \text{ET } (\text{point} \quad \text{EXTERIEUR} \quad \text{triangle})^* \end{array} \right.$
$\longrightarrow \left\{ \begin{array}{l} B \equiv (\text{triangle}, \text{ cercle}) \\ \text{ET } (\text{triangle A-GAUCHE} \quad \text{ cercle}) \\ \text{ET } (\text{triangle EXTERIEUR} \quad \text{ cercle}) \\ \text{ET } (\text{ECHELLE triangle} \quad = 2) \\ \text{ET } (\text{ECHELLE } \text{ cercle} \quad = 1) \end{array} \right.$

Une dernière relation subsiste dans C : c'est « point EXTERIEUR cercle », elle s'identifie directement à la deuxième relation de A. Si l'on applique à l'expression C le théorème $T \equiv (A \rightarrow B)$, c'est-à-dire les mêmes transformations qui permettent de passer

de A à B, on obtient donc la description D qui est B transformée par les substitutions qui ont permis d'unifier A et B, c'est-à-dire a/point , $b/\text{ cercle}$, $c/\text{triangle}$.

D \equiv (triangle, cercle)
ET (triangle A-GAUCHE cercle)
ET (triangle EXTERIEUR cercle)
ET (ECHELLE triangle=2)
ET (ECHELLE cercle=1)

Ici encore, les échelles sont repérées par rapport à la figure d'origine, soit C dans ce cas. Il ne reste plus qu'à voir s'il est possible d'identifier cette description avec l'une des figures proposées en réponse. **Ceci, à nouveau, est du ressort de l'unification**, puisqu'il s'agit d'identifier deux descriptions (sans tenir compte de l'ordre des relations). Cette dernière étape est simplement rendue triviale, puisqu'en fait, aucune substitution n'est autorisée, à cause de la remarque déjà faite sur les « variables » de C. La bonne réponse existe, c'est le troisième dessin à droite de la figure 3.17.

Notons que aurions pu gagner du temps dans cette recherche en tenant compte du fait que A et C ne contiennent chacun qu'une seule relation du type « $x \text{ INCLUS-DANS } y$ ». Commençant à unifier ces relations, on obtient : c changé en triangle et b en cercle, puis nécessairement : a changé en point et le résultat.

3.9.2 Commentaires sur ce type de problèmes de tests géométriques

Même si nous n'en sommes pas toujours très conscients, je pense que nous nous servons implicitement de la procédure d'unification pour répondre à ce type de tests.

Quelques questions liées à cette démarche subsistent cependant. Tout d'abord l'oeil humain - avec tout le système visuel - est un merveilleux outil de traitement de l'information : le passage du dessin aux descriptions géométriques telles qu'elles ont été données nous est facile et familier. Il n'en est actuellement pas du tout de même pour un ordinateur muni d'une ou deux caméras : c'est tout le problème de la « *reconnaissance des formes* ».

En second lieu, il n'est pas évident, contrairement à ce qui était implicitement supposé dans le paragraphe précédent, qu'il existe une description standard des objets et des relations. Ainsi :

« triangle A-GAUCHE cercle » peut bien sûr aussi s'écrire :
 « cercle A-DROITE triangle ».

Le même phénomène apparaît à chaque fois qu'une relation admet une relation inverse : INTERIEUR et EXTERIEUR, DESSOUS et DESSUS. Ceci entraîne des modifications à l'intérieur de l'algorithme d'unification pour détecter toutes les substitutions possibles. Or, pour peu que les expressions soient longues, le nombre de descriptions équivalentes - qui croît de manière exponentielle - peut être très grand.

Lié à ce point, qui concerne donc l'efficacité de l'algorithme d'unification et la pertinence des descriptions, apparaît le problème de *l'information parasite* : on peut toujours décrire plus finement les figures dans notre langage symbolique ; le stade ultime correspond en fait au tracé exact des dessins sur le papier. Ainsi on peut ajouter à la description de la figure A le fait que le point n'est pas seulement extérieur au triangle,

mais est, en plus, à sa gauche et en haut. Dans C le point est, au contraire, à droite et au milieu. On voit qu'alors l'unification ne peut se faire exactement. Pour qu'elle réussisse il faut se replacer dans la description que nous avons effectivement utilisée auparavant. Les informations décrivant plus précisément la position relative des points est considérée alors comme non-pertinente, ce qui est ici justifié a posteriori par le fait que les points de A et de C disparaissent de la description finale dans B et dans D.

Une troisième remarque concerne le **langage** de description lui-même. Celui-ci n'est **pas donné**, il n'est pas commun à tous les individus comme peut l'être le langage mathématique usuel. Il n'est pas aussi rigoureux, ni aussi bien formalisé. Ainsi le vocabulaire des relations entre objets doit être effectivement étendu pour prendre en compte d'autres caractéristiques :

Le TYPE-DU-TRAIT (continu ou tirets ou pointillés...);

La COULEUR (blanc ou noir ou grisé...);

L'ORIENTATION (par rapport à un axe de symétrie par exemple).

Mais, moyennant toutes ces extensions de langage, des figures plus complexes peuvent de fait être traitées automatiquement. Par exemple, la série des tests de QI de Cattell (figure 3.18).

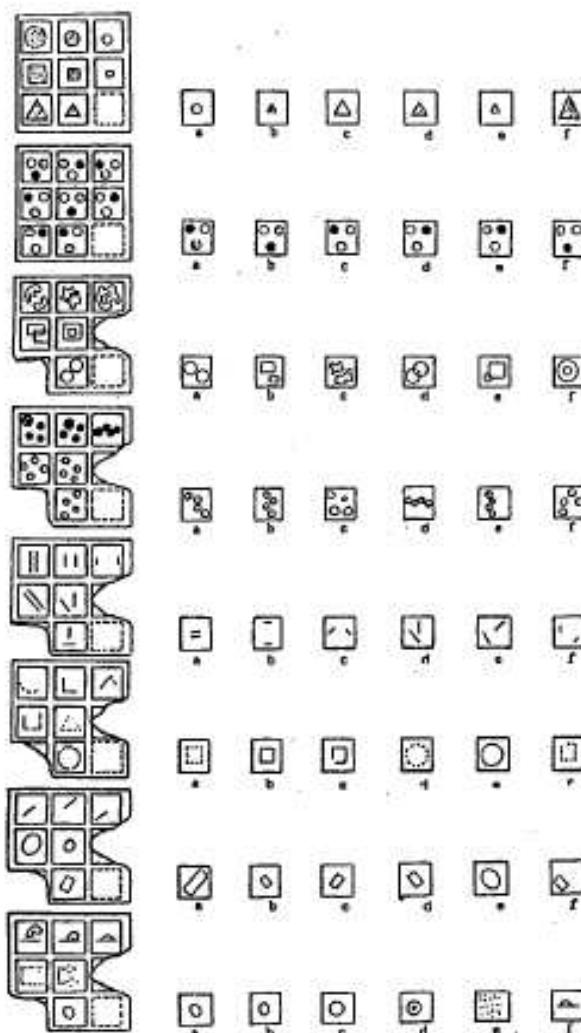


FIG. 3.18 – Tests de Catell

Les tests qui portent sur des suites de nombres, de lettres ou de dominos sont souvent plus difficiles dans la mesure où :

- Aucune famille de réponses possibles n'est habituellement fournie ;
- Plusieurs unifications sont en général admissibles, donc plusieurs réponses logiquement envisageables.

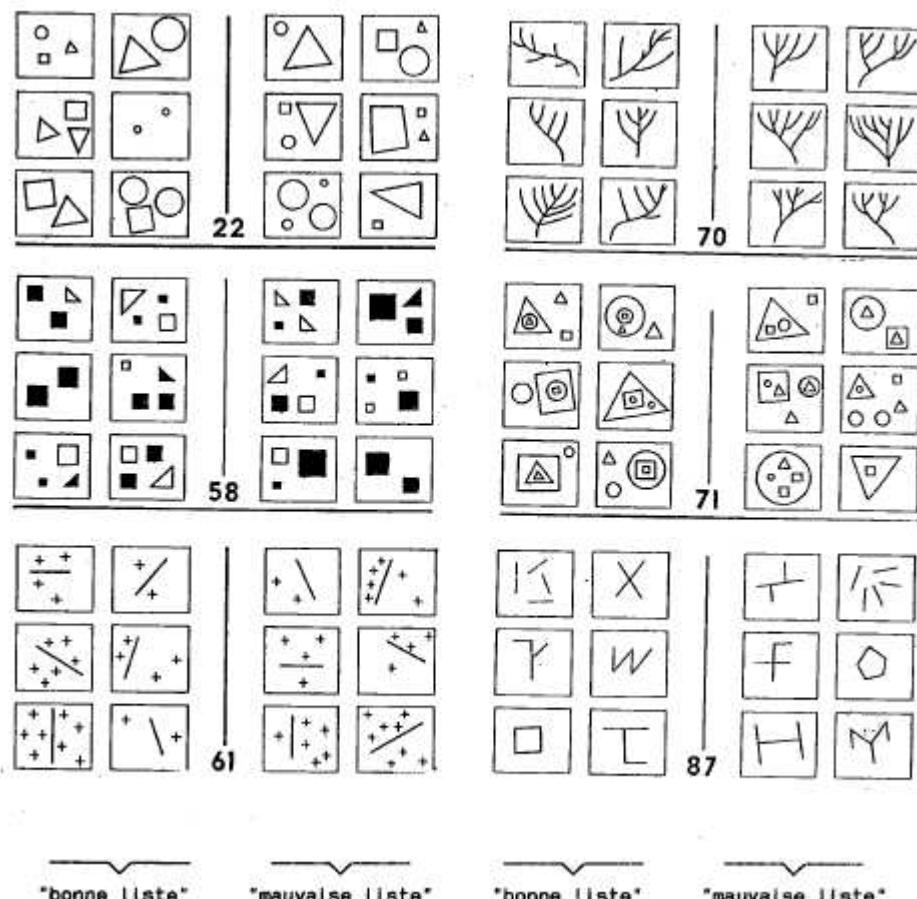


FIG. 3.19 – Quelques problèmes de R. Bongard

Trouver le concept qui partage pour chaque groupe de 12 figures la « bonne liste » à gauche, de la « mauvaise liste » à droite.

Exemples :

3	7	16	35	74	?
U	D	T	Q	C	?
1	4	7	11	14	17 41 ?
691	602	513	424	335	?
2	4	5	6	4	3 4 4 4 ?
13	50	27	26	55	14 111 8 ?
B	E	I	N	T	?
123456789?					

C'est la raison pour laquelle tout un ensemble de données est toujours proposé. Il prend la forme : $U_1 \rightarrow U_2 \rightarrow U_3 \rightarrow \dots U_i \rightarrow ?$ On espère ainsi limiter l'ambigüité en donnant la possibilité de filtrer progressivement les transformations envisageables :

on ne retient que celles qui fonctionnent à *toutes* les étapes de U_1 à U_i . Il n'en reste pas moins qu'il faut finalement choisir entre plusieurs possibilités. En effet, il passe toujours une infinité de courbes par un nombre fini de points (les points jouent ici le rôle des items donnés, les courbes sont les lois de transformation des items). Les tests du mathématicien russe R. Bongard (figure 16) sont dans le même cas. C'est principalement pourquoi ces tests de mesure du « Quotient Intellectuel » ont été et sont encore si critiqués. Il faut en effet introduire au bout du compte un critère de « simplicité » - tout à fait *subjectif* - pour départager les transformations licites.

3.9.3 Utilisation de l'algorithme d'unification en démonstration automatique

Nous sommes d'ores et déjà en mesure d'écrire un programme efficace pour démontrer quelques théorèmes dans des formalismes simples : il pourra servir plus tard de banc d'essai pour tester des programmes sophistiqués, comme par exemple ceux écrits à partir du Principe de Résolution (paragraphe II 1.9) ou des systèmes experts (chapitre VII). Une seule question subsiste pour faire concrètement tourner sur machine l'algorithme d'unification : **Il faut éliminer les parties localement infinies de l'arbre de recherche.**

3.9.4 La méthode de Siklóssy et Marinov

Certaines règles de réécriture donnent, en particulier, **un membre droit sur lequel la même règle peut à nouveau être appliquée**. De telles règles seront dites *expansives*, elles engendrent évidemment un ensemble infini de formules et leur emploi doit être limité d'une manière ou d'une autre. Par exemple, en logique des propositions, la règle : $p \mapsto \neg\neg p$ est expansive. A l'aide de l'algorithme d'unification lui-même, le programme peut bien sûr se rendre compte si une règle fournie en donnée est ou non expansive.

Les règles du type : $x+y \mapsto y+x$ qui, appliquées deux fois consécutivement sur la même expression, laissent celle-ci inchangée, ne peuvent provoquer une recherche infinie, et ne sont que *pseudo-expansives*. Laurent Siklóssy et Vladimir Marinov proposent de ne considérer les règles expansives que lorsqu'on ne peut absolument rien faire d'autre. Elles sont alors déclenchées au plus une fois pour le niveau de recherche courant, et la recherche complète sur toutes les autres règles est relancée. Ils obtiennent, en 1971, de brillants résultats par rapport à ce que nous pouvons faire à la main. Donnons deux exemples de démonstration avec cette simple heuristique, indépendante du domaine, **de mise à l'écart des règles expansives**.

Exemple 1 : Logique des propositions.

Six règles sont données

- $R_1 : p \wedge q \mapsto q \wedge p$
- $R_2 : p \wedge (q \wedge r) \mapsto (p \wedge q) \wedge r$
- $R_3 : p \wedge (p \supset q) \mapsto q$
- $R_4 : \neg q \wedge (p \supset q) \mapsto \neg p$
- $R_5 : \neg\neg \mapsto p$
- $R_6 : p \mapsto \neg\neg p$

et le théorème à démontrer est proposé :

$$T : (s \supset \neg t) \wedge t \mapsto \neg s$$

La règle expansive R6 est ôtée dans un premier temps et la recherche se déroule de manière systématique : les expressions déjà rencontrées ne sont pas à nouveau développées et sont **marquées mortes**. La conclusion du théorème à démontrer ne sert que comme **test d'arrêt** : la recherche a lieu à l'aveuglette.

L'arbre de recherche est celui de la figure 3.20 où les noeuds ont été numérotés dans l'ordre d'obtention, c'est-à-dire en **largeur d'abord** : toutes les dérivations possibles, à un même niveau, pour une même expression, sont effectuées avant de passer au niveau suivant.

Seule la règle R1 peut s'unifier avec l'expression de départ H (noeud 1). Elle redonne ensuite, bien sûr, H elle-même, et R2, R3, R4 et R5 ne peuvent toujours pas s'appliquer. La règle R6 est alors lancée **pour un niveau**. Elle s'unifie de trois façons différentes avec H, pour donner trois expressions (noeuds 2, 3 et 4), sur lesquelles R1 s'applique, suivie de R4 qui donne le succès pour le noeud 7 (figure 3.20).

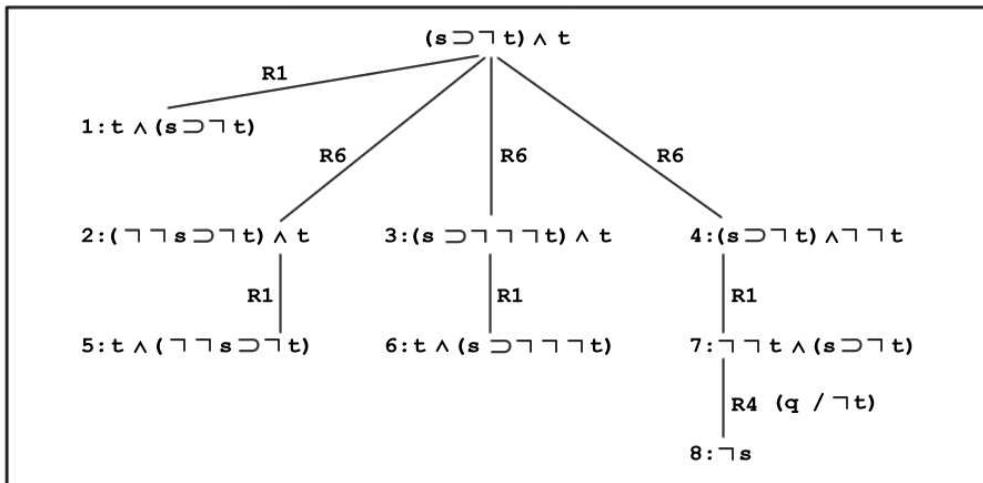


FIG. 3.20 – Arbre de dérivation de : $(s \supset \neg t) \wedge t \mapsto \neg s$

Dans cette démonstration, toutes les unifications, par souci de simplicité, n'ont pas été effectuées : en premier lieu les lettres propositionnelles de l'hypothèse du théorème à démontrer n'ont pas été considérées comme des variables substituables (semi-unification), en second lieu l'application de la règle R6 à l'expression $(s \supset \neg t) \wedge t$ dans son entier n'a pas été considérée.

Exemple 2 : Théorie élémentaire des groupes.

Six règles sont à nouveau données :

- $S_1 : a + b \mapsto b + a$
- $S_2 : a + (b + c) \mapsto (a + b) + c$
- $S_3 : (a + b) - b \mapsto a$
- $S_4 : a \mapsto (a + b) - b$
- $S_5 : (a - b) + c \mapsto (a - b) + c$
- $S_6 : (a + b) - c \mapsto (a - c) + b$

Elles doivent permettre de démontrer le théorème :

$$T_1 : (a + b) + c \mapsto a + (b + c)$$

Tentez vous-même cette démonstration. Comme l'algèbre donnée ici est restrictive par rapport à ce que nous connaissons, nos stratégies habituelles sont en échec et nous évoluons péniblement. La règle S_1 est repérée comme pseudo-expansive par le système : par contre la règle S_4 est repérée comme expansive. Par ailleurs, le système fera un usage implicite du métathéorème suivant : tant que S_4 n'est pas déclenchée, la connective $-$ n'apparaît pas et seules S_1 et S_2 peuvent s'appliquer. La solution est obtenue en 11 pas en n'utilisant que S_1 et S_2 : elle est donnée par la figure 3.21.

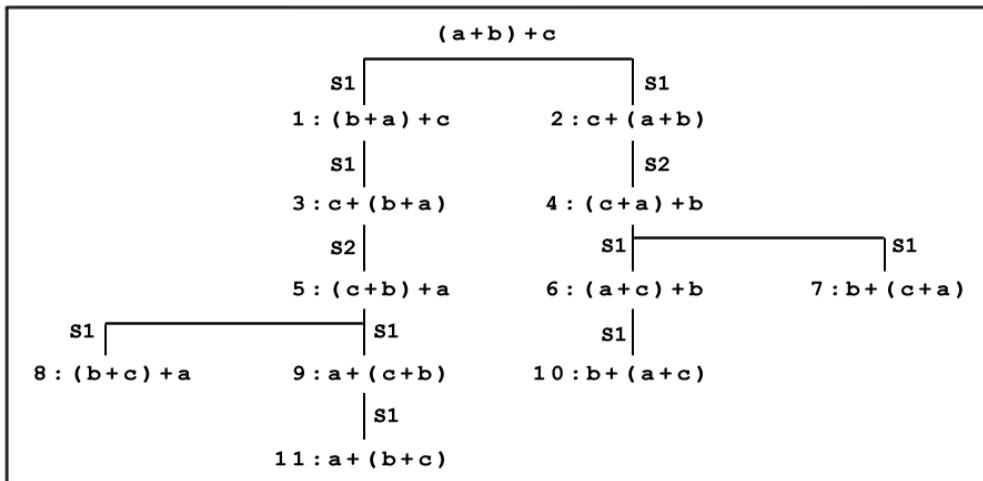


FIG. 3.21 – Arbre de dérivation de $(a + b) + c \mapsto a + (b + c)$

Avec les mêmes règles, le théorème plus intéressant :

$$T2 : (a + c) - (b + c) \mapsto a - b$$

demande 351 pas. mais seulement quelques secondes d'unité centrale... !

Le programme que nous venons de décrire ne peut certes pas prétendre démontrer un jour des résultats mathématiques importants. Notons cependant qu'il peut, fondamentalement, contrairement à plusieurs autres systèmes, trouver des théorèmes nouveaux : Il suffit de le laisser tourner. Ce programme met bien en évidence la **puissance de manipulation symbolique d'un ordinateur** : les plus longues démonstrations demandent quelques secondes. Il constitue donc un bon sous-programme pour des systèmes où la preuve symbolique n'est pas le sujet principal (par exemple l'écriture automatique de programmes), ou bien encore pour démontrer des résultats intermédiaires dans un domaine qui n'est pas le domaine essentiel du système de démonstration : un mathématicien spécialisé en topologie sait faire appel à ses livres d'analyse pour faire un calcul qu'il ne connaît pas.

Une amélioration de la méthode énumérative de Siklóssy et Marinov consiste à conserver la répartition en deux classes de règles : les règles « obligatoires », qui correspondent en fait à une normalisation-simplification des expressions d'une part, les règles « non obligatoires » (en particulier expansives), d'usage « dangereux », d'autre part. Elle consiste, par ailleurs, à affiner le processus de déclenchement de ces dernières. Nous donnons ci-après les descriptions de deux démonstrateurs de théorèmes qui participent de ces idées.

3.9.5 Résolution d'équations trigonométriques : PRET (M. Grandbastfen (1974)).

Le système reçoit des énoncés de la forme : $P(\text{trigo}(x)) = \text{constante}$ où $P(\text{trigo}(x))$ désigne un polynôme des arcs trigonométriques (\sin , \cos , \tan , \cotan) de la variable x . Pour résoudre ces équations, le programme connaît les solutions des deux « cas du cours » :

$$\begin{cases} \text{trigo}(x) = c & (B_1) \\ a \cos x + b \sin x = c & (B_2) \end{cases}$$

Notons qu'on aurait pu songer à se ramener à une résolution d'équation en t , où $t = \tan \frac{x}{2N}$, N désignant le PGCD des dénominateurs dans P , ou bien encore au changement : $t = e^{ix/2N}$, par passage aux formules d'Euler. Mais dans les deux cas le degré de l'équation résultante est, en général, trop élevé pour autoriser une solution algébrique.

Ces changements de variables conviennent par contre pour démontrer des *identités* trigonométriques et non plus résoudre des équations. Quand il s'agit d'identités il suffit, en effet, de montrer qu'un polynôme est identiquement nul, ce que l'on sait faire quel que soit le degré.

La méthode utilisée par le système PRET va consister à tenir compte des buts -(B_1) ou (B_2)- et à tout faire pour s'en rapprocher. La démarche générale comprend une itération sur trois étapes :

- Appliquer les règles obligatoires tant que cela est possible ;
- Appliquer une règle non obligatoire convenablement choisie ;
- Sélectionner, parmi l'ensemble des expressions en cours d'étude, la plus prometteuse.

Chacun des points 1 à 3 va être précisé. Commençons par le point 3. Par construction, la recherche est arborescente et de chaque expression naissent plusieurs filles. Si chacune de ces expressions filles a été engendrée par une règle différente à partir de l'expression mère, la filiation est étiquetée d'une marque OU : pour résoudre l'équation de départ, il faut et il suffit de résoudre l'une des équations-filles équivalentes (figure 3.22).

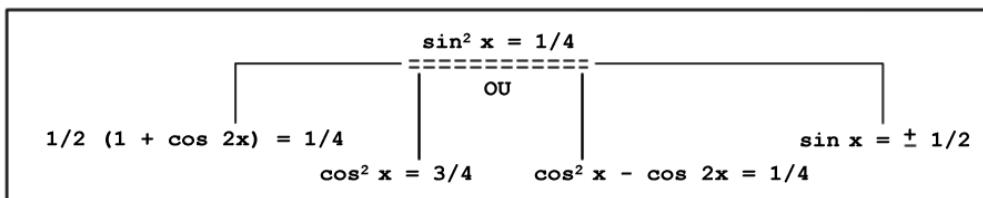


FIG. 3.22 – Noeud OU en démonstration

Si l'expression-mère a été mise sous forme d'un produit de facteurs, et que le membre de droite est 0, il faut résoudre séparément l'équation relative à chacun des facteurs. Autant de sous-problèmes sont alors engendrés, qui sont reliés par la marque ET.

Enfin, en toute étape de la résolution., il convient de choisir l'expression que le programme va traiter. Ceci se fait à l'aide d'une fonction d'évaluation sur les filles du

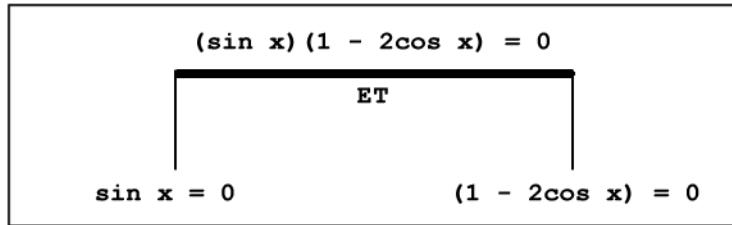


FIG. 3.23 – Noeud ET en démonstration (on veut toutes les solutions)

dernier noeud OU : PRET retient l'expression dont la complexité est la moins élevée. Pour toute expression e la complexité $C(e)$ est donnée par :

$$\begin{aligned} C(e) &= \text{nombre de facteurs au premier niveau} \\ &+ \text{nombre d'arguments différents} \\ &+ \text{somme des puissances différentes de l'unité.} \end{aligned}$$

Cette évaluation est très simple, mais elle est suffisante car $C(e)$ augmente très vite si l'expression e est mal choisie, ce qui provoque l'abandon de la branche. PRET passe à un autre noeud e lorsque toutes les évaluations des expressions filles sont devenues trop grandes.

Au point 2, le programme lance les règles non obligatoires (RNO). Celles-ci sont **ordonnées** : il les applique - ce sont des règles de réécriture - partout où c'est possible. La liste complète de ces règles est donnée de façon abrégée par le tableau (RNO) (tableau 3.3) où trigo désigne un arc trigonométrique quelconque. L'ensemble des règles non obligatoires a été déterminé expérimentalement à contrario : ce sont les règles qu'il n'est pas toujours bon d'appliquer en trigonométrie, soit parce qu'elles compliquent l'expression : c'est le cas de (g), soit parce qu'elles éloignent du but : (c), soit parce qu'elles forment des combinaisons expansives : (i) et (j).

Finalement, le point 1 se divise en réalité en trois points : en premier lieu, le programme PRET tente, si cela est possible, de couper le problème en sous-problèmes par **mise en facteurs**, et création d'un noeud ET. Sinon, PRET tente de simplifier l'expression par un **changement de variable** : il se ramène éventuellement à un seul argument ou à un seul arc trigonométrique. Si tout cela échoue, l'expression est standardisée en appliquant par unification les règles obligatoires (RO) dans l'ordre donné. Elles sont classées en 10 groupes (tableau 3.4).

Dans le cas où un but doit être atteint, une table indique, pour la suite, les seules transformations à retenir pour ce but. Plus de 100 exercices ont été résolus par PRET en quelques secondes de Cil 10070. Donnons deux exemples :

Exemple 1 :

$$\sqrt{3} \cdot \tan x - 4 \sin^2 x = 0 \quad (\text{I})$$

Aucune mise en facteur, aucun changement de variable standard n'est possible. En revanche, la (RO)-f, se déclenche. L'expression devient :

$$\sqrt{3} \cdot \frac{\sin x}{\cos x} - 4 \cdot \sin^2 x = 0 \quad (\text{II})$$

Alors la (RO)-j donne, si $\cos x \neq 0$:

$$\sqrt{3} \cdot \sin x - 4 \cdot \sin^2 x \cdot \cos x = 0 \quad (\text{III})$$

a)	$\text{trigo } u \pm \text{trigo } v$	$\mapsto 2\text{trigo } \left(\frac{u+v}{2}\right) \text{trigo } \left(\frac{u-v}{2}\right)$
b)	$\sin 2x$	$\mapsto 2 \sin x \cos x$
	$\cos 2x$	$\mapsto 2 \cos^2 x - 1$
c)	$\sin u \sin v$	$\mapsto \frac{1}{2} (\cos(u-v) - \cos(u+v))$
	$\cos u \cos v$	$\mapsto \frac{1}{2} (\cos(u-v) + \cos(u+v))$
d)	$\text{trigo } (u+v)$	$\mapsto f(\text{trigo } u, \text{trigo } v)$
e)	$\text{trigo}^\alpha u$	$\mapsto f(\text{trigo } \alpha u)$
	$\sin^2 u$	$\mapsto \frac{1}{2}(1 + \cos 2u)$
	$\cos^3 u$	$\mapsto \frac{1}{4}(\cos 3u + 3 \cos u)$
f)	$\text{trigo } (-u)$	$\mapsto f(\text{trigo } u)$
	$\text{trigo } (\pi \pm u)$	$\mapsto f(\text{trigo } u)$
	$\text{trigo } \left(\frac{\pi}{2} \pm u\right)$	$\mapsto f(\text{trigo } u)$
	$\text{trigo } \left(\frac{\pi}{4} \pm u\right)$	$\mapsto f(\text{trigo } u)$
g)	$\text{trigo } (2x)$	$\mapsto f(\tan u)$
h)	$\sin^2 x$	$\mapsto 1 - \cos^2 x$
	$\sin x \cos x$	$\mapsto \frac{1}{2} \sin 2x$
i)	$\tan u$	$\mapsto \frac{\sin u}{\cos u}$
j)	$\frac{\sin u}{\cos u}$	$\mapsto \tan u$
h)	$\cos^2 u$	$\mapsto \frac{1}{1+\tan^2 u}$

TAB. 3.3 – Tableau (RNO) – Règles non obligatoires en trigonométrie.

- a) Développement (règle de réécriture) :
 $(a + b)^n \mapsto$ somme
- b) Suppression des radicaux sur l'expression dans son ensemble (règle de production) :
 $\sqrt{a} = c \rightarrow a = c^2$
- c) Somme donne produit sur l'expression dans son ensemble :
 $\sin u + \sin v \mapsto 2 \sin\left(\frac{u+v}{2}\right) \cos\left(\frac{u-v}{2}\right)$
- d) Mise en facteurs (règle de réécriture) :
 $a^n - b^n \mapsto (a - b) \cdot (\text{polynôme})$
- e) Elimination des cotan, si l'expression n'a qu'un seul argument :
 $\cotan x \mapsto \frac{1}{\tan x}$
- f) Elimination des tan, si l'expression n'a qu'un seul argument :
 $\tan x \mapsto \frac{\sin x}{\cos x}$
- g) Si sin et cos apparaissent avec des puissances impaires division par $\cos^\alpha x$, but :
 $(P(\tan x) = 0) \vee (P(\cos x) = 0)$.
- h) Si sin et cos apparaissent avec des puissances paires :
 $\sin^2 x \mapsto 1 - \cos^2 x$, but $P(\tan x) = 0$
- i) Si l'expression e est symétrique en sin et cos :
 $x \mapsto x + \frac{\pi}{4}$
- j) S'il existe des dénominateurs :
 Réduire au dénominateur commun.

TAB. 3.4 – Tableau (RO) – Régles obligatoires (RO) en trigonométrie

L'expression ayant été transformée. PRET revient au point 1 et la mise en facteur de $\sin x$ est détectée :

$$\sin x (\sqrt{3} - 4 \cdot \sin x \cdot \cos x) = 0$$

Cette expression donne lieu à deux problèmes marqués ET :

$$\sin x = 0, \text{ qui est trivial,} \quad (\text{IV})$$

$$\text{et : } \sqrt{3} - 4 \cdot \sin x \cdot \cos x = 0 \quad (\text{V})$$

qui devient la nouvelle expression courante. Les règles obligatoires ne donnent plus rien, mais la (RNO)-h s'applique :

$$\sin x \cos x \mapsto \frac{1}{2} \sin 2x$$

donc, l'expression devient :

$$\sin 2x = \frac{\sqrt{3}}{2} \quad (\text{VI})$$

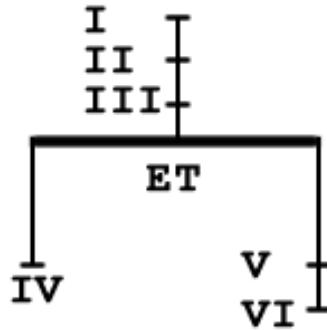


FIG. 3.24 – Arbre de résolution de : $\sqrt{3} \cdot \tan x - 4 \sin^2 x = 0$

Exemple 2 :

$$\sin(n+1)x + \sin(n-1)x - \sin 2x = 0 \quad (\text{I})$$

Aucune (RO) ne se déclenche sur cette expression I. La (RNO)-a donne par contre naissance à (II) :

$$2 \sin nx \cdot \cos x - \sin 2x = 0 \quad (\text{II})$$

dont la complexité est devenue : $4 + 3 + 0 = 7$, alors que la complexité de l'expression (I) valait : $3 + 3 + 0 = 6$.

En réalité, la même règle (RNO)-a s'applique aussi pour regrouper, non plus les termes 1 et 2, mais aussi bien les termes 1 et 3 ou 2 et 3. PRET engendre donc simultanément :

$$2 \sin \frac{n-1}{2}x \cdot \cos \frac{n+3}{2}x + \sin(n-1)x = 0 \quad (\text{III})$$

et :

$$2 \sin \frac{n-3}{2}x \cdot \cos \frac{n+1}{2}x + \sin(n+1)x = 0 \quad (\text{IV})$$

Les expressions (III) et (IV) ont aussi la même complexité : 7. Maintenant PRET considère la (RNO)-b, qui se déclenche sur l'expression d'origine, pour donner :

$$\sin(n+1)x + \sin(n-1)x - 2 \sin x \cos x = 0 \quad (\text{V})$$

dont la complexité est $5 + 3 + 0 = 8$.

Aucune RNO ne s'applique plus et PRET redémarre sur la première expression de complexité minimale, soit (II). La (RNO)-b s'applique bien sûr alors à nouveau, et PRET obtient l'expression (VI) :

$$2 \sin nx \cdot \cos x - 2 \sin x \cos x = 0 \quad (\text{VI})$$

sur laquelle une mise en facteur est immédiate :

$$2 \cos x (\sin nx - \sin x) \quad (\text{VII})$$

Elle donne naissance à deux sous-problèmes (VIII) et (IX). La (RO)-c convertit la somme (IX) en produit :

$$\sin nx - \sin x = 2 \sin \frac{n-1}{2} x \cos \frac{n+1}{2} x \quad (\text{X})$$

et le problème est ainsi résolu dans son entier (figure 3.25).

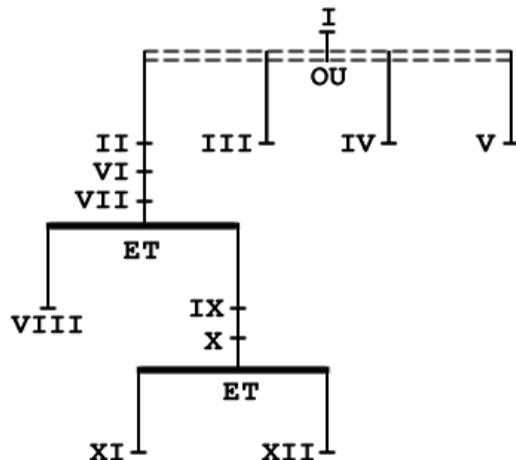


FIG. 3.25 – Arbre de résolution de $\sin(n+1)x + \sin(n-1)x - \sin 2x = 0$

3.9.6 Résolution d'exercices d'arithmétique : PARI (D. Bourgoin (1978))

La connaissance mathématique dans PARI est organisée autour de modules spécifiques. L'idée générale, pour résoudre un problème, est de viser l'un des problèmes précis que l'on sait résoudre et de tout faire pour s'y ramener. Pour PARI, qui accepte n'importe quel énoncé portant sur des variables entières, six modules savent traiter des sous-problèmes déterminés. Ils jouent le rôle de **règles obligatoires évoluées**.

<Module 1> : « **produit = constante** ». Ce module utilise la décomposition d'un nombre en facteurs premiers et les critères de divisibilité.

Exemple : $x \cdot y = 5 \Rightarrow (x = 5, y = 1)$ OU $(y = 5, x = 1)$

<Module 2> : « **congruence** ». Il raisonne sur des équations simples en se basant sur les congruences des petits nombres premiers : 2, 3, 5, 7, 11, de manière purement combinatoire. Il déduit ainsi des conditions nécessaires que doivent satisfaire les inconnues.

Exemple : $5x = 2y + 1 \Rightarrow x$ impair

<Module 3> : « **divisibilité** ». Il raisonne à partir du lemme fondamental de GAUSS : « Si a est premier avec b et divise le produit bc , alors a divise c ».

Exemple : 3 divise $7z \Rightarrow 3$ divise z

<Module 4> : « **équation** ». Il sait trouver des solutions particulières à des équations simples par essais successifs (l'idéal, ici, serait l'utilisation des fractions continues).

Exemple : $7x - 5y = 11$ a pour solution particulière $x = 3$ et $y = 2$

<Module 5> : « **division euclidienne généralisée** ». Il calcule formellement le quotient et le reste de deux polynômes.

Exemple : $x^2 - 2x + 1$ divisé par $x - 3$ donne

$$x^2 - 2x + 1 = (x - 3)(x + 1) + 4$$

<Module 5> : « **récurrence** ». Il sait trouver, à partir d'une valeur initiale et d'une relation de récurrence, l'expression formelle du terme général.

Exemple : $u_n = 3u_{n-1} + 4$; $u_1 = 5 \Rightarrow u_n = 3^{n-1}(5a + 2) - 2$

Ces modules utilisent des *programmes de service* pour normaliser les expressions, effectuer des substitutions, reporter les résultats trouvés dans les relations courantes. Le but final recherché est de donner explicitement la solution, soit sous forme exhaustive, soit sous forme d'une condition simple la caractérisant, soit enfin de donner la démonstration de son inexistence.

A chaque étape, l'**adéquation** de chacun des modules au problème en cours est jugée au moyen d'un **jeu de six triplets**. Chacun d'entre eux mesure, pour chacun des six modules, la pertinence du module, sa difficulté d'application (elle tient compte du nombre de variables et des degrés maximum) et enfin, du coût résultant attendu pour la suite du problème. Des critères précis permettent alors de décider quel triplet retenir et donc quel module appliquer.

Exemples de résolutions :

Exemple 1 : « Déterminer deux nombres premiers p et q tels que :

$$p^2 = 8q + 1$$

Le module « divisibilité » est celui qui cadre le mieux avec la condition imposée. Il considère qu'il a affaire à une équation en q et pose la nouvelle condition de divisibilité :

$$8|(p^2 - 1) \quad (8 \text{ divise } p^2 - 1)$$

Le module « congruence » reprend celle-ci, mais n'obtient rien de neuf (p et q sont déjà réputés impairs, puisque premiers). Par contre, ce même module devient maintenant le candidat le mieux placé pour étudier directement la condition de départ et les congruences modulo 3 donnent naissance à :

$$(p \text{ modulo } 3)^2 \equiv 2(q \text{ modulo } 3) + 1$$

Trois cas de congruences sont étudiés successivement :

SI $p \equiv 0$ modulo 3, alors p , qui est premier, ne peut qu'être égal à 3 :

il vient : $9 = 8q + 1$ soit $q = 1$;
on obtient donc une solution : $p = 3, q = 1$

SI $p \equiv 1$ modulo 3, l'équation de congruence devient :

$1 \equiv 2(q \text{ modulo } 3) + 1$ soit $q = 0$ (modulo 3),
ou bien $q = 3$, puisque q doit être premier ;
d'où : $p^2 = 8.3 + 1$ et $p = 5$ mais ceci contredit l'hypothèse : $p \equiv 1$ modulo 3.

SI $p \equiv 2$ modulo 3, il vient dans ce dernier cas :

$4 \equiv 2 (q \text{ modulo } 3) + 1$ soit à nouveau : $q \equiv 0$ (modulo 3) et $q = 3$, d'où $p^2 = 25$
et $p = 5$ qui, ici convient.

Le problème admet donc deux solutions (3, 1) et (5, 3).

Exemple 2 : « Trouver les entiers positifs x et n tels que :

$$\frac{5x^3 - 6x^2 + 7x + 8}{x^2 - 3x + 4} = n$$

Le module « division euclidienne », qui reçoit la meilleure note pour ce problème, est lancé, il produit l'identité :

$$5x^3 - 6x^2 + 7x + 8 \equiv (x^2 - 3x + 4)(5x + 9) + 14x - 28$$

Le trinôme $(x^2 - 3x + 4)$ doit diviser les deux membres de cette identité, il doit donc diviser le « reste » : $(14x - 28)$, le module « divisibilité » forme la nouvelle condition :

$$x^2 - 3x + 4 | 14x - 28$$

que reprend le module « division euclidienne » pour former la nouvelle identité :

$$14(x^2 - 3x + 4) \equiv (14x - 28)(x - 1) + 28$$

A nouveau $(x^2 - 3x + 4)$ doit diviser les deux membres de cette identité, en particulier : $x^2 - 3x + 4 | 28$, soit :

$$k(x^2 - 3x + 4) = 28, \text{ avec } k \text{ entier.}$$

Le module « produit = constante » prend ici le relai pour tester les valeurs $k = \pm 1, \pm 2, \pm 4, \pm 7, \pm 14, \pm 28$, PARI effectue la résolution des 12 équations du second degré correspondantes.

Par exemple : $k = +14$ conduit à $x^2 - 3x + 4 = 2$ soit $x = 1$ et $x = 2$.
 $k = +7$ conduit à $x^2 - 3x = 0$ soit $x = 0$ et $x = 3$

L'ensemble des solutions est alors 0, 1, 2, 3, 5. Mais le raisonnement que vient d'effectuer PARI n'est pas, et il le sait, une suite d'équivalences, mais une suite de conditions nécessaires. Il convient de tester chaque solution sur l'expression de départ ; 3 est ainsi éliminé et il ne reste que les solutions 0, 1, 2 et 5 pour ce problème.

Exemple 3 : « Trouver tous les entiers n tels que :

$$\sum_{i=1}^n 2i = n(n+1) \quad \text{(I)}$$

Le module « récurrence » est appelé. Il vérifie d'abord l'égalité au rang 1 : $2 = 1(1+1)$: elle est triviale. Il substitue alors, dans (I), $(n+1)$ à n pour tenter de démontrer la propriété au rang $(n+1)$ à partir de la propriété au rang n . Le terme général à l'ordre $(n+1)$ est isolé :

$$\sum_{i=1}^n 2i + 2(n+1) \stackrel{?}{=} (n+1)((n+1)+1)$$

Utilisant alors (I) comme une règle de réécriture, cette conjecture devient :

$$n(n+1) + 2(n+1) = (n+1)(n+2),$$

soit, après mise en facteur et simplification par $(n+1)$ qui est par hypothèse un entier positif différent de -1 : $n+2 \stackrel{?}{=} n+2$, égalité triviale. Le théorème initial est ainsi démontré par récurrence.

Quelques problèmes résolus par PARI :

Trouver a et b tels que $a|b$ et $a|(b+l)$. Démontrer les identités :

$$\sum_{i=1}^n (2i-1)^2 = \frac{n(2n-1)(2n+1)}{3}$$

$$\sum_{i=1}^n 4(i+1)(i+3)(i+5) = n(n+1)(n^2+13n+52) + 60n$$

$$\sum_{i=1}^n i^3 = \left(\sum_{i=1}^n i \right)^2$$

$$nr^2(n+1)(2n+1) + 6arn(n+1) + 6na^2 - 6 \sum_{i=1}^n (a+ir)^2 = 0$$

Soit la suite définie par :
$$\begin{cases} u_1 = 3a + 2 \\ u_{n+1} = 3u_n + 2 \end{cases}$$

Montrer que le $n^{\text{ème}}$ terme de cette suite peut s'écrire : $u_n = ab^n$;

Résoudre dans \mathbb{Z} les équations définies par :

$$2x^3 + xy + 7 = 0$$

$$p(100 - q) + q(100 - p) = 1978$$

$$x^2 + 8z = 3 + 2y^2$$

Pour quelles valeurs de x : $x^2 + 8kx - 3k^2$ divise-t-il $4x^3 + 35kx^3 + 5k^2x^2 - 65k^3x + 21k^4$?

Démontrer les congruences suivantes : $(10k + 7)^{10} + 1 \equiv 0(10)$
 $(6k + l).2^{6k+1} + 1 \equiv 0(3)$

Montrer que 5^{2n} et 2^{5n} ont mêmes restes de division par 7 ;

Trouver x tel que : $5|4x^2 + 1$ et $13|4x^2 + 1$.

Démontrer que si p et $8p^2 - 1$ sont premiers, $8p^2 - 1$ est composé.

Démontrer que l'équation : $p^2 + q^2 = r^2 + s^2 + t^2$ n'admet pas de solution avec p, q, r, s, t , premiers supérieurs à 2.

Déterminer tous les entiers n ($n \in \mathbb{N}$) tels que les 6 nombres $n + 1, n + 3, n + 7, n + 9, n + 13, n + 15$, soient premiers.

Tous les programmes précédents se contentaient d'obtenir la démonstration du théorème proposé ou la solution du problème donné. La démarche du mathématicien est toute autre : il ne connaît pas à l'avance les théorèmes qu'il cherche, il n'a aucun critère d'arrêt a priori. Ce qui guide le mathématicien, c'est son » intuition » ou cette espèce de connaissance de niveau supérieur qui lui permet à la fois de diriger ses recherches et de simplifier son travail en court-circuitant des étapes de détail sans intérêt. Autrement dit, il est guidé par une **métathéorie**.

Il existe des programmes qui utilisent effectivement une métathéorie. Nous décrivons ci-après le premier d'entre eux, dû à Jacques Pitrat (1966).

3.10 Le programme de J. PITRAT en logique des propositions

3.10.1 Hiérarchie des métathéories

Un théorème est un énoncé d'un système formel. Un métathéorème est un énoncé sur les théorèmes d'un système formel. Son intérêt est double :

- il permet, d'une part, de **raccourcir** les démonstrations en évitant de répéter les intermédiaires classiques. De plus, autorisant le passage direct d'une arborescence comme celle gauche dans la figure 3.26. à la dérivation linéaire de droite :
- il donne, d'autre part, une meilleure **vision globale** de la recherche, il permet une **focalisation** sur les choix importants sans obscurcir la mémoire de résultats partiels sans intérêt.

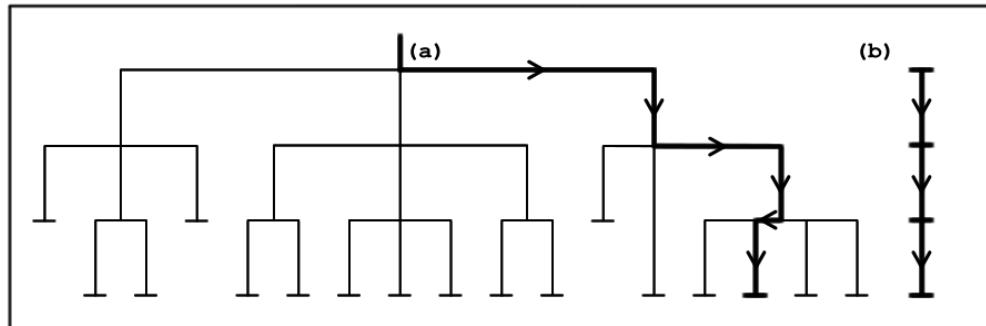


FIG. 3.26 – Arbre de dérivation (a) et dérivation linéaire (b)

Les métathéorèmes peuvent être manipulés à l'aide d'un second S.F. dans lequel on prend soin de distinguer, à l'aide de notations différentes, les nouveaux opérateurs. Ainsi le modus ponens est une règle de dérivation qui permet d'engendrer des théorèmes, c'est donc aussi un métathéorème qu'on écrit :

$$p, p \supset q \rightarrow q$$

« Si p est théorème et si $p \supset q$ est un théorème, alors q est un théorème. La « , » et la « \rightarrow » sont les pendants, dans la métalangue, du \wedge et du \supset

Les métathéorèmes peuvent être démontrés à l'intérieur d'un S.F. Ainsi nous avons démontré que, dans (LP), de tout théorème de la forme :

$$P \supset (Q \supset R)$$

on pouvait déduire, par modus ponens et substitutions, le nouveau théorème :

$$Q \supset (P \supset R)$$

Ce résultat, qui est un métathéorème de (LP), soit : $P \supset (Q \supset R) \rightarrow Q \supset (P \supset R)$, s'applique à l'évidence à de nombreux énoncés de (LP). Un moyen commode pour construire différemment des métathéorèmes est de faire usage de *meta-métathéorèmes*.

Par exemple :

« Si de a on peut déduire b et si de b on peut déduire c , alors on peut déduire c de a » est un métamétathéorème qui exprime la transitivité de la déduction, propriété qui est valable dans maintes théories. Pour pouvoir l'écrire, il nous faut de nouveaux symboles pour l'implication et pour la conjonction dans la métalangue. Nous choisissons le « ; » et la « \multimap ». Le résultat précédent s'énonce alors :

$$a \rightarrow b; b \rightarrow c \multimap a \rightarrow c$$

Il paraît légitime de donner de tels métamétathéorèmes **généraux**, utilisés par tout mathématicien, à un programme. Malheureusement, il existe aussi des métamétathéorèmes forts importants et néanmoins spécifiques d'un S.F. donné. Ainsi :

$$p \supset q \multimap p \rightarrow q \tag{3.7}$$

n'est pas légal dans toutes les théories logiques. Il fait en effet intervenir un opérateur particulier de la langue elle-même : le symbole \supset , celui-ci n'existe pas nécessairement dans toutes les théories. D'une façon générale, tous les métamétathéorèmes où figurent des opérateurs du S.F. d'origine sont spécifiques. Il faut donc se donner un moyen de démontrer des métamétathéorèmes. On définit donc une *méta-méta-métathéorie*. A ce niveau, le degré de généralité atteint est tel qu'il est aujourd'hui valable pour toutes les mathématiques. Ainsi, l'énoncé : « SI $a, b \rightarrow c$ est un métathéorème, alors $a \multimap b \rightarrow c$ est un métamétathéorème » constitue un métaméta-métathéorème universellement admis.

3.10.2 Les métathéories du programme :

Le programme de J. Pitrat travaille dans différentes axiomatiques de la logique des propositions, il utilise huit métamétathéorèmes généraux qui sont :

-
- 1) $a, b \rightarrow c \rightarrow b, a \rightarrow c$ (commutativité des antécédents)
 2) $a, a \rightarrow b \rightarrow a \rightarrow b$ (simplification)
 3) $v \rightarrow a \rightarrow a$ (où v est une variable ne figurant pas dans a)
 4) $v, a \rightarrow b \rightarrow a \rightarrow b$ (où v est une variable ne figurant pas dans a)
 5) $a; a \rightarrow b \rightarrow b$ (modus ponens de la métathéorie)
 6) $a \rightarrow b; b \rightarrow c \rightarrow a \rightarrow b$ (transitivité de la déduction 1ère forme)
 7) $a \rightarrow b; b, c \rightarrow d \rightarrow a, c \rightarrow d$ (transitivité de la déduction 2ème forme)
 8) $a \rightarrow b; c, d \rightarrow a \rightarrow c, d \rightarrow b$ (transitivité de la déduction 3ème forme)

Ces mèta-métaméorèmes, bien que non explicités, sont utilisés par les mathématiciens. En outre, pour pouvoir démontrer d'autres mèta-métathéorèmes spécifiques d'une théorie, le programme connaît cinq mèta-méta-métathéorèmes. Le « ! » désigne ici la conjonction, et la « $\circ\circ\rightarrow$ » désigne la déduction dans la mèta-méta-métalangue :

- I) $a, b \rightarrow c \circ\circ\rightarrow a \rightarrow b \rightarrow c$
 II) $a, b \rightarrow c \circ\circ\rightarrow v \rightarrow a \rightarrow v, b \rightarrow c$
 (la variable v ne figure ni dans a ni dans b ni dans c)
 III) $a, b \rightarrow c \circ\circ\rightarrow c \rightarrow v \rightarrow a, b \rightarrow v$
 IV) $a! a \rightarrow b \rightarrow c \circ\circ\rightarrow b \rightarrow c$
 V) $a \rightarrow b! a \rightarrow b \rightarrow c, d \rightarrow e \circ\circ\rightarrow c, d \rightarrow e$

	Langue	M-langue	M-M-langue	M-M-M-langue
ET	\wedge	,	;	!
Implication	\supset	\rightarrow	$\rightarrow\circ\circ\rightarrow$	$\circ\circ\rightarrow$

FIG. 3.27 – Notations dans les différentes langues

Ces énoncés permettent d'établir rapidement des résultats importants dans les S.F. rencontrés. Ils sont utilisés par le programme, via l'algorithme d'unification, qui se chargera de faire coïncider les différents (meta)-théorèmes obtenus avec les membres gauches ci-dessus. Ainsi, lorsque dans une théorie particulière, le mot :

$$p, q \rightarrow p \wedge q$$

aura été démontré, (III) permettra d'en déduire le mèta-métathéorème :

$$(p \wedge q \rightarrow r) \rightarrow\circ\circ\rightarrow (p, q \rightarrow r)$$

Deux questions fondamentales doivent de plus être résolues par le programme pour posséder un comportement inventif :

- Comment choisir les résultats intéressants ?
- Comment choisir les essais à tenter ?

Choix des résultats :

Par utilisation des meta, mèta-méta, et mèta-méta-métathéorèmes, le programme est vite confronté à une explosion combinatoire des éléments à archiver.

Trois idées générales sont programmées dans le but d'éviter celle-ci.

- a) Un énoncé est d'autant plus intéressant qu'il comporte moins de symboles ;
- b) Une production est d'autant plus intéressante que l'antécédent est proche du conséquent. Rappelons qu'une production est une règle de dérivation, c'est-à-dire un (meta)*-théorème avec un ou plusieurs antécédents.
- c) Mieux vaut passer du temps à décider un archivage que d'en perdre, plus tard, en vain essais.

Les **productions** sont traitées à part car la philosophie à leur égard est simple : il en faut très peu. Il faut prendre des mesures sévères pour ne pas stocker des versions différentes d'une forme plus générale, pour ne pas dupliquer par transitivité des résultats connus : si on a déjà $a \rightarrow b$ et $b \rightarrow c$, il n'est pas bon de mémoriser aussi $a \rightarrow c$, car le programme retrouvera facilement cette étape naturelle. Par contre les dérivations qui mènent à un **résultat simple après une étape plus compliquée**, sont systématiquement conservées.

L'intérêt d'une production est toujours évalué en fonction de ce qui est déjà connu et des énoncés qu'elle est potentiellement à même de produire : tel résultat rejeté au début de l'étude peut être, plus tard, archivé : tel résultat considéré comme très intéressant dans un système donné peut être éliminé dans un autre. Le programme n'archive que les productions les plus simples à un instant donné : ce sont celles qui contiennent le moins d'opérateurs et le moins de variables propositionnelles.

Le cas des **théorèmes** est plus délicat puisque, si les productions sont par définition formées d'antécédents et d'un conséquent que l'on peut comparer, un théorème ne comporte qu'une suite de symboles de la langue. *L'intérêt d'un théorème*, pour le programme, est mesuré par la somme des intérêts des productions qu'il permet d'engendrer, augmentés d'une fonction linéaire décroissante du nombre de ses opérateurs unaires d'une part, binaires d'autre part. Mais en réalité, l'intérêt des théorèmes est ailleurs : il est lié aux **interprétations** possibles du S. F. La liaison créée entre langue et métalangue dans l'examen des productions issues d'un théorème, ne traite qu'en partie ce problème.

3.10.3 Choix des essais :

On ne considère d'abord que le résultat d'intérêt le plus grand, au sens indiqué ci-dessus. Si c'est un théorème, on lui applique toutes les productions à un antécédent qui *simplifient* (qui augmentent l'intérêt). Si c'est une production qui simplifie, on l'applique à tous les théorèmes archivés et à toutes les productions qui compliquent en prenant celles-ci comme antécédents. Si c'est une production qui complique, on l'applique aux productions qui simplifient ainsi qu'aux productions à plusieurs antécédents. Tous les essais possibles ne sont pas effectués, car le programme archive, avec chaque énoncé, les combinaisons auxquelles celui-ci a déjà pris part et ne les recommence pas.

On voit que la sélection des essais est sommaire, mais, comme les résultats ne seront ensuite pas tous archivés, le temps machine et la place mémoire restent dans des limites acceptables.

Résultats.

Le programmme a travaillé sur six théories différentes dues à Russell, Lukasiewicz, Hilbert et Bernay, Sheffer. Il retrouve tous les théorèmes importants et en donne quelquefois des démonstrations originales. Il fait usage de tous les métaniveaux qui lui sont donnés. Il sait en outre fonctionner en mode conjecture : cette fois un énoncé précis lui est soumis et il doit le démontrer, ses stratégies sont alors quelque peu différentes puisqu'il inverse ses productions pour remonter de la conjecture vers les axiomes. Dans tous les cas, le programme réussit à démontrer lui-même des théorèmes que son programmeur n'était pas parvenu à prouver. Un jugement impartial est celui de Lukasiewicz : « One must be very expert in performing such proofs. If we want to deduce from the three axioms of logic the law of commutation : $(p \supset (q \supset r)) \supset (q \supset (p \supset r))$, or

even the law of simplification : $p \supset (q \supset p)$ ». Le programme de J. Pitrat démontre effectivement ces deux résultats dans l'axiomatique en question.

Un exemple de démonstration

Axiomes :	$(p \supset (q \supset r)) \supset ((p \supset q) \supset (p \supset r))$	T_1
	$p \supset (q \supset p)$	T_2
	$(\neg p \supset \neg q) \supset (q \supset q)$	T_3
Production :	$p, (p \supset q) \supset q$	M_1

La production M_1 donne d'abord naissance à deux méta-métathéorèmes grâce aux (meta)*-théorèmes dont le programme dispose. Par commodité, nous utiliserons, dans la démonstration, la notation du programme - la polonaise préfixée - pour les théorèmes et pour eux seulement. Le programme obtient :

$$\begin{aligned} & \supset pq \multimap p \rightarrow q \quad MM_1 \text{ par } M_1 \text{ et I)} \\ \text{et : } & p \rightarrow \supset qr \multimap p, qr \quad MM_2 \text{ par } M_1 \text{ et II)} \end{aligned}$$

Par exemple, MM_2 se déduit du méta-méta-métathéorème II) et de M_1 , avec $p \supset q$ pour a , p pour b , et q pour c en tenant compte de la commutativité du ', donné par 1). Il vient ainsi :

$$\begin{aligned} (v \rightarrow \supset p) & \multimap (v, p \rightarrow q) \text{ soit, en normalisant :} \\ p \rightarrow \supset qr & \multimap p, q \rightarrow r, \text{ qui est } MM_2 \end{aligned}$$

Donnons maintenant la partie de l'arbre de dérivation qui conduit ici finalement au théorème remarquable : $(\neg p \supset p) \supset p$. Pour y parvenir, de façon entièrement autonome, le programme construit 5 méta-métathéorèmes, 21 métathéorèmes et 20 théorèmes selon les procédures décrites ci-dessus. Notons les résultats importants démontrés en cours de route : $T_4, T_7, T_8, T_9, T_{11}, T_{12}, T_{13}, T_{16}$. D'autres théorèmes ont été démontrés dans cette axiomatique et dans d'autres : le lecteur en trouvera le détail dans Pitrat (1966).

3.11 Le Principe de Résolution et le Langage PRO-LOG

3.11.1 Le Principe de Résolution

En 1930, Jacques Herbrand passe à Paris sa thèse de mathématique. Il y expose une méthode originale de démonstration de théorèmes en logique du premier ordre. L'idée générale est la suivante : pour démontrer une conclusion C à partir des hypothèses H_1, H_2, \dots, H_n , soit pour démontrer le théorème T :

$$H_1 \wedge H_2 \wedge \dots \wedge H_n \supset C$$

Il peut être plus facile de démontrer que la formule F :

$$H_1 \wedge H_2 \wedge \dots \wedge H_n \wedge \neg C \tag{F}$$

où la négation de la conclusion est ajoutée aux hypothèses, est contradictoire. Pour cela, il suffit en effet de montrer que F est un non-théorème, comme contenant une sous expression de la forme $p \wedge \neg p$; autrement dit, il suffit d'exhiber un contre-exemple pour la formule (F). Herbrand définit un processus effectif qui atteint ce but par un

T_2	+	MM_1	$p \rightarrow \supset qp$	M_2
T_3	+	MM_1	$\supset \neg p \neg q \rightarrow \supset qp$	M_3
T_1	+	MM_1	$\supset p \supset qr \supset \supset pq \supset pr$	M_4
M_4	+	MM_2	$\supset p \supset qr, \supset pq \rightarrow \supset pr$	M_5
			La production M_5 donne naissance à un MMThéorème :	
			$\supset p \supset qr, \supset pq \rightarrow \supset pr$	MM_3
T_2	+	MM_3	$\supset pq \rightarrow \supset pp$	M_6
T_2	+	M_6	$\supset pp$	T_4
M_2	+	premier	antécédent de M_5 $\supset pq, \supset rp \rightarrow \supset rq$	M_7
			La production M_7 donne naissance aux MMThéorèmes :	
			$\supset pq \rightarrow \supset qr \rightarrow \supset pr$	MM_4
			$\supset pq \rightarrow \supset rp \rightarrow \supset rq$	MM_5
T_3	+	MM_5	$\supset p \supset \neg q \neg r \rightarrow \neg p \neg rq$	M_8
T_2	+	MM_4	$\supset \supset pqr \rightarrow \supset qr$	M_9
M_4	+	M_9	$\supset p \supset qr \rightarrow \supset q \supset qr$	M_{10}
T_4	+	M_{10}	$\supset p \supset \supset pqq$	T_5
T_5	+	M_4	$\supset \supset p \supset pq \supset pq$	T_6
T_6	+	MM_1	$\supset p \supset pq \rightarrow \supset pq$	M_{11}
T_6	+	MM_5	$\supset p \supset q \supset qr \rightarrow \supset p \supset qr$	M_{12}
T_1	+	M_9	$\supset \supset pq \supset \supset rp \supset rq$	T_7
T_7	+	MM_1	$\supset pq \rightarrow \supset \supset rp \supset rq$	M_{13}
T_7	+	M_{10}	$\supset \supset pq \supset \supset qr \supset pr$	T_8
T_8	+	MM_1	$\supset pq \rightarrow \supset \supset qr \supset pr$	M_{14}
T_2	+	M_{14}	$\supset \supset \supset pqr \supset qr$	T_9
T_9	+	MM_5	$\supset p \supset \supset qrs \rightarrow \supset p \supset rs$	M_{15}
T_1	+	M_{15}	$\supset \supset p \supset qr \supset q \supset pr$	T_{10}
T_{10}	+	MM_4	$\supset \supset p \supset qrs \rightarrow \supset \supset q \supset prs$	M_{16}
T_3	+	M_9	$\supset \neg p \supset pq$	T_{11}
T_{11}	+	MM_4	$\supset \supset pqr \rightarrow \supset \neg pr$	M_{17}
M_{14}	+	M_{17}	$\supset pq \rightarrow \supset \neg q \supset r$	M_{18}
M_{18}	+	M_8	$\supset \neg pq \rightarrow \supset \neg q \supset rp$	M_{19}
M_{19}	+	M_{11}	$\supset pq \rightarrow \supset \neg qp$	M_{20}
T_4	+	M_{20}	$\supset \neg \neg pp$	T_{12}
T_{12}	+	M_3	$\supset p \neg \neg p$	T_{13}
T_{13}	+	M_{13}	$\supset \supset pq \supset p \neg \neg q$	T_{14}
T_{14}	+	M_8	$\supset \supset \neg pq \supset \neg qp$	T_{15}
T_{15}	+	M_{13}	$\supset \supset p \supset \neg qr \supset p \supset \neg rq$	T_{16}
T_{16}	+	M_{12}	$\supset \supset \neg p \supset \neg qp \supset \neg pq$	T_{17}
T_{17}	+	M_{16}	$\supset \supset \neg p \supset \neg qq \supset \neg qp$	T_{18}
T_{18}	+	MM_1	$\supset \neg p \supset \neg qq \rightarrow \supset \neg qp$	M_{21}
T_{12}	+	M_{21}	$\supset \neg p \neg \supset \neg pp$	T_{19}
T_{19}	+	M_3	$\supset \supset \neg ppp$	T_{20}

TAB. 3.5 – Démonstration du théorème $(\neg p \supset p) \supset p$

nombre fini de substitutions dans (F) , **à chaque fois que T est initialement un théorème.** Les paragraphes suivants précisent ce processus.

3.11.2 Théorème d'Herbrand

La démonstration de la contradiction de F repose sur l'interprétation du symbole de négation, \neg . Le théorème d'Herbrand va assurer que la contradiction peut toujours être atteinte en un nombre fini de pas, quelles que soient les valeurs de vérité données aux fonctions qui interviennent dans les hypothèses et la conclusion. Définitions : Décidons tout d'abord de travailler uniquement sur des expressions standard, disjonctives. C_i , appelées *clauses* :

$$C_i = (L_{i_1} \vee L_{i_2} \vee \dots \vee L_{i_{n_i}} \vee)$$

F s'écrivant alors comme une conjonction de clauses : $F = \wedge C_i$. Nous verrons un peu plus loin qu'une telle standardisation est toujours possible pour une formule quelconque du premier ordre. Les quantités L_{j_k} , appelées *littéraux*, ne contiennent aucun des symboles \wedge, \vee, \exists ou \supset .

Soit alors E l'ensemble des clauses. G_0 l'ensemble des constantes apparaissant dans E et, par itération, pour tout $i \in \mathbb{N}$, G_i l'ensemble formé par la réunion de G_{i-1} et de tous les termes $f_k(t_1, t_2, \dots, t_{m_k})$, où f_k est une fonction à m_k arguments des termes t_j ($j = 1, 2, \dots, m_k$, t_j terme de G_{i-1}).

On pose :
$$G(E) = \bigcup_{i=0}^{\infty} G_i$$

L'ensemble $G(E)$ est, par définition, l'univers de Herbrand de E . Par ailleurs, la *base de Herbrand*, $B(E)$, d'un ensemble de clauses E est l'ensemble de toutes les *formules atomiques* de E , c'est-à-dire comportant au plus un prédicat ou un prédicat précédé du symbole \neg . La démonstration du théorème de Herbrand consiste à construire une arborescence A , sur les éléments de $B(E)$, telle que :

- a) E est la racine de A ;
- b) Tout sommet de A est un sous-ensemble éventuellement instantié de E ;
- c) Tout chemin issu de la racine ne contient pas à la fois p et $\neg p$, où p est une formule quelconque.

Associer une valeur de vérité aux formules construites sur $B(E)$ revient alors à choisir un chemin dans A . La contradiction sera obtenue lorsque la condition 3) ci-dessus ne pourra plus être satisfaite. Le théorème de Herbrand assure qu'il en est toujours ainsi quand E est inconsistant (contradictoire).

Théorème de Herbrand (1930) : Une condition nécessaire et suffisante pour qu'un ensemble e de clauses soit inconsistant, est qu'il existe un ensemble fini contradictoire de sous-formules de E complètement instanciées.

Cet ensemble peut être construit par une arborescence du type précédent à l'aide d'un procédé très simple, lié au résultat fondamental suivant :

Théorème de Résolution (1930) : Soit la formule, sous forme standard, F_1 :

$$F_1 = C_1 \wedge C_2 \wedge \dots \wedge (p \vee L_i) \wedge \dots \wedge (\neg p \vee L_j) \wedge \dots \wedge C_m$$

et soit la formule F_2 :

$$F_2 = F_1 \wedge (L_i \vee L_j).$$

Si F_2 est contradictoire, alors F_1 est également contradictoire.

Démonstration :

Il suffit de montrer que $F_1 \rightarrow F_2$. Le cas où $L_i = \emptyset$ est, tout d'abord, trivial, puisqu'il suffit alors de montrer que si $E_2 = L_j$ est contradictoire, alors :

$$E_1 = p \wedge (\neg p \vee L_j)$$

l'est aussi, les C_k , pour $k \neq i$ et $k \neq j$, n'intervenant pas. Or, en logique du premier ordre, on a :

$$\begin{aligned} (\neg A \vee B) &\equiv (A \supset B) \\ \text{donc : } E_1 &= p \wedge (p \supset L_j) \end{aligned}$$

soit encore, par modus ponens et le métathéorème $p, q \leftrightarrow p \wedge q : E_1 = L_j$, et le résultat est acquis, puisque $F_1 \rightarrow F_2$. Le cas inverse du précédent, où $L_j = \emptyset$ se démontre de la même façon, en remarquant que $p \equiv \neg \neg p$:

$$\begin{aligned} E_1 &\equiv (\neg p \vee L_i) \wedge \neg p \\ \text{soit } E_1 &\equiv (\neg \neg p \vee L_i) \wedge \neg p = (\neg p \supset L_i) \wedge \neg p \\ \text{donc, à nouveau : } E_1 &\equiv L_i \text{ et } F_1 \rightarrow F_2 \end{aligned}$$

Dans le cas général, E_1 prend la forme :

$$E_1 \equiv (p \vee L_i) \wedge (\neg p \vee L_j)$$

Il vient comme précédemment :

$$\begin{aligned} p \vee L_i &\rightarrow \neg L_i \supset p \text{ et } \neg p \vee L_j \rightarrow p \supset L_j \\ \text{donc : } E_1 &\equiv (\neg \neg p \vee L_i) \wedge \neg p = (\neg p \supset L_i) \wedge \neg p \\ \text{et par application du théorème : } (a \supset b) \wedge (b \supset c) &\rightarrow (a \supset c) \end{aligned}$$

$$\begin{aligned} E_1 &\rightarrow \neg L_i \supset L_j \\ \text{soit encore : } L_i \vee L_j & \\ \text{et donc : } F_1 &\rightarrow F_2 \end{aligned}$$

Ainsi F_2 est déductible de F_1 , la clause $(L_i \vee L_j)$ n'ajoute rien et si F_2 est contradictoire, c'est que F_1 l'était déjà. De plus, le processus de « Résolutions » successives ainsi établi, est *complet*, c'est-à-dire que si une formule E est contradictoire, la Résolution démontrera cette contradiction en un nombre fini d'étapes ; c'est ce qu'assure le théorème de J. Herbrand.

Les clauses $C_i = (p \vee L_i)$ et $C_j = (\neg p \vee L_j)$ sont appelées *clauses parentes* de la clause, dite *clause résolvante*, $L_i \vee L_j$. Bien entendu, si les clauses C_i et C_j contiennent des variables, ce qui est le cas général en logique du premier ordre, c'est par **unification** que sera recherché un littéral commun p . Plusieurs unifications peuvent donner ce littéral, de telle sorte que c'est bien une **arborescence de formules** qu'il faut construire pour trouver la démonstration, c'est-à-dire la contradiction.

Exemples de Résolutions :

- Logique des propositions : Soit la formule $F_1 = \neg s \wedge q \wedge (p \vee \neg q) \wedge (\neg p \vee s)$ formée de quatre clauses en conjonction $C_1 \wedge C_2 \wedge C_3 \wedge C_4$. Une première résolution entre C_1 et C_4 conduit à la résolvante $C_5 \equiv \neg p$. Une seconde résolution entre C_2 et C_3 donne $C_6 \equiv p$. Cette clause C_6 unifiée avec la clause C_5 donne une résolvante vide : la formule de départ F_1 était donc contradictoire. Autrement dit nous venons d'établir, en ajoutant le négation $\neg s$ aux hypothèses, le théorème : $q \wedge (p \vee \neg q) \wedge (\neg p \vee s) \supset s$.
- Logique du premier ordre : Soient x, y, z, t , et v , cinq variables, a, b et c trois constantes. P, Q et R trois prédictats quelconques et l'expression F_1 :

$$F_1 \equiv P(x, y) \wedge Q(t) \wedge R(v) \wedge (\neg P(a, z) \vee Q(b) \vee \neg Q(c))$$

L'unification fournit trois substitutions et trois résolvantes possibles à partir de F_1 :

$$\begin{array}{lll} S & = & \{x/ay/z\} \text{ donne : } F_2 & = & \neg Q(b) & \vee \neg Q(c) \\ S' & = & \{t/b\} \text{ donne : } F'_2 & = & \neg P(a, z) & \vee \neg Q(c) \\ S'' & = & \{t/c\} \text{ donne : } F''_2 & = & \neg P(a, z) & \vee \neg Q(b) \end{array}$$

Ici encore, la démonstration s'achèvera lorsque L_i et L_j seront trouvés tous deux vides, c'est-à-dire lorsqu'une contradiction interne sera exhibée pour une résolvante. C'est le cas pour F_2 et la seconde clause de F_1 . La substitution :

$$\begin{array}{ll} S''' & = \{t/b\} \text{ donne } F_3 = \neg Q(c) \text{ qui donne à nouveau avec la même clause, et} \\ S'''' & = \{t/c\} \text{ la résolvante : } F_4 = (\text{clause vide}), \text{ ainsi } F_1 \text{ était contradictoire.} \end{array}$$

3.11.3 Organisation pratique des démonstrations par le principe d'Herbrand

Si les clauses sont standardisées en conjonctions, il devient inutile de conserver les symboles \wedge . Pour gérer l'arborescence, il suffit, dès lors, de conserver ces clauses les unes derrière les autres. Lors d'une résolution, la clause résolvante est ajoutée en haut d'une pile, sans destruction des clauses parentes, de façon à conserver les autres possibilités de substitutions (figure 3.28). Après la substitution S , on place en haut de pile l'élément $\neg Q(b) \vee \neg Q(c)$.

$\neg P(a, z) \vee \neg Q(b) \vee \neg Q(c)$
$R(u)$
$Q(t)$
$P(x, y)$

FIG. 3.28 – Mémorisation de l'arborescence de démonstration

Une seule occurrence de chaque clause est conservée. En outre, si une clause résolvante C_1 est moins générale qu'une clause C_2 déjà dans la pile, elle n'est pas conservée : si il existe une substitution S telle que $S.C_2 = C_1$ alors on rejette C_1 . Ce serait le cas de $C_1 = Q(a)$, avec $C_2 = Q(t)$. Notons, de plus, que l'ordre des clauses dans la pile n'a pas d'importance à cause de la commutativité de l'opérateur A .

Notons que le nombre de littéraux d'une résolvante est égal à la somme des nombres de littéraux de ses parents diminuée de deux unités : Il **augmente**, sauf si l'un des deux parents a, au plus, deux littéraux. La contradiction n'étant finalement trouvée qu'avec deux clauses parentes *unitaires*, c'est-à-dire à un seul littéral.

3.11.3.1 Mise sous forme normale conjonctive.

Il reste à montrer maintenant comment mettre toute expression de la logique du premier ordre sous forme conjonctive normalisée. Six étapes définissent cet algorithme de transformation :

- 1) **Mise sous forme « prénexé » :** Cette étape consiste à faire passer en tête tous les quantificateurs. Cette opération, qui éloigne les symboles des variables qu'ils doivent contrôler, ne nous est pas très naturelle. Il n'y a pas de commutativité possible et

l'ordre des quantificateurs doit absolument être conservé. Il y a lieu, de plus, de renommer éventuellement les variables quantifiées lors de ce processus en utilisant les théorèmes :

$$(Q_1 x) A(x) * (Q_2 x) B(x) \rightarrow (Q_1 x) (Q_2 y) (A(x) * B(y))$$

où Q_1 désigne soit \forall soit \exists et $*$ désigne soit \vee , soit \wedge . De même :

$$(\forall x) [P(x) \supset (\exists x) Q(x)] \rightarrow (\forall x) [P(x) \supset (\exists y) Q(y)]$$

Exemple : Si la formule initiale E est par exemple :

$$\forall x (P(x) \supset ((\exists y) (P(y) \vee \neg R(a, x, y)) \supset (V z) (\neg S(y, z))))$$

elle est devenue après cette phase :

$$\forall x (\exists y) (\forall z) (P(x) \supset ((P(y) \vee \neg R(a, x, y)) (\neg S(y, z))))$$

2) Elimination des quantificateurs existentiels :

Cette idée est due au mathématicien Skolem (1927). Elle consiste à remarquer que toute variable quantifiée existentiellement est en réalité une fonction des variables quantifiées universellement dont elle dépend, c'est-à-dire dans la portée desquelles elle se trouve. A chacune d'elle est donc substituée, par *skolémisation*, une nouvelle fonction, et les quantifications existentielles sont ôtées. Notre expression E devient ainsi avec $y = f(x)$:

$$\forall x (\forall z) (P(x) \supset ((P(f(x)) \vee \neg R(a, x, f(x))) (\neg S(f(x), z))))$$

3) Elimination des quantificateurs universels :

Toutes les variables qui subsistent sont maintenant quantifiées universellement. Les quantificateurs peuvent devenir implicites. E s'écrit :

$$(P(x) \supset ((P(f(x)) \vee \neg R(a, x, f(x))) (\neg S(f(x), z))))$$

Si à ce point, il reste des quantificateurs existentiels seuls en tête, on les supprime également : il convient ensuite de trouver des installations des variables associées qui satisfont la formule.

4) Elimination des symboles \supset et \leftrightarrow :

On utilise les théorèmes : $A \leftrightarrow B \mapsto (A \supset B) \wedge (B \supset A)$
et : $A \supset B \mapsto \neg A \vee B$

autant de fois qu'il est nécessaire. Notre formule devient :

$$(\neg P(x) \vee (\neg(P(f(x)) \vee \neg R(a, x, f(x))) \vee \neg S(f(x), z)))$$

Trois opérateurs seulement sont donc maintenant utilisés pour toute formule de (PP) : \vee , \wedge , et \neg

5) Réduction de la portée des négations :

Les théorèmes :

$$\neg(A \vee B) \rightarrow (\neg A \wedge \neg B) \text{ et } \neg(A \wedge B) \rightarrow (\neg A \vee \neg B)$$

permettent de ne faire progressivement porter les signes \neg que sur des prédicts uniques. E s'écrit :

$$(\neg P(x) \vee ((\neg P(f(x)) \wedge R(a, x, f(x))) \vee \neg S(f(x), z)))$$

6) Passage en forme conjonctive :

Cette ultime étape a pour but de mettre partout l'opérateur \wedge en tête de façon à se ramener à une formule du type :

$$C_1 \wedge C_2 \wedge \dots \wedge C_i \wedge \dots \wedge C_n,$$

où les clauses C_i ne contiennent que les connectives \vee et \neg . On utilise pour cela les théorèmes de distributivité :

$$A \vee (B \wedge C) \rightarrow (A \vee B) \wedge (A \vee C)$$

$$\text{et } (A \wedge B) \vee C \rightarrow (A \vee C) \wedge (B \vee C)$$

E s'écrit :

$$(\neg P(x) \vee ((\neg P(f(x)) \vee \neg S(f(x), z)) \wedge (R(a, x, f(x)) \vee \neg S(f(x), z))))$$

La pile des clauses avant Résolution, en ôtant les parenthèses devenues inutiles dans les expressions disjonctives, est ainsi celle de la figure 3.29.

$\neg P(x) \vee R(a, x, f(x)) \vee \neg S(f(x), z)$
$\neg P(x) \vee \neg P(f(x)) \vee \neg S(f(x), z)$

FIG. 3.29 – Formé normale conjonctive et pile de résolution

Notons que, dans l'algorithme 1) à 6), ci-dessus présenté, certaines étapes peuvent être permuteées sans fausser le résultat ; les quantificateurs, par exemple, peuvent n'être ôtés qu'en tout dernier lieu.

C'est J. A. Robinson qui le premier, en 1963, a rendu les idées de J. Herbrand opérationnelles sur ordinateur.

3.11.4 Exemples de démonstrations suivant le principe d'Herbrand :

Donnons d'abord un exemple complet de démonstration que pouvait obtenir le système de Robinson et notons son caractère inhabituel : supposons que nous voulions montrer **qu'« une main est une partie d'un homme »**.

Nous avons besoin d'un prédictat binaire A ; $A(x, y)$ signifiant : x fait partie de y , et de l'expression qui exprime la transitivité de A . soit $A(x, y) \wedge A(y, z) \supset A(x, z)$, traduite ici en forme normale disjonctive : $\neg A(x, y) \vee \neg A(y, z) \vee +A(x, z)$ où $\neg P$ remplace dorénavant la suite de symboles $\neg P$, et $+P$ le symbole P pour tout prédictat P , ce qui permet d'omettre les symboles \vee dans les clauses (notation PROLOG).

Les hypothèses seront, si m, b, t et h sont quatre constantes qui désignent respectivement une main, un bras, un torse et un homme : $A(m, b)$, $A(b, t)$ et $A(t, h)$, auxquelles

est ajoutée la négation de la conclusion : $\neg A(m, h)$. Le programme démarre donc avec cinq clauses implicitement reliées par des \wedge :

C_1	$-A(x, y) - A(y, z) + A(x, z)$
C_2	$+A(m, b)$
C_3	$+A(b, t)$
C_4	$+A(t, h)$
C_5	$\neg A(m, h)$

Nous donnons ici les seules résolutions qui permettent d'aboutir le plus vite à une contradiction. L'arborescence effective trouvée par le programme comporte bien d'autres clauses et bien d'autres chemins. On effectue successivement les résolutions :

Clauses parentes avec les numéros des littéraux unifiés	Substitutions	Résolvantes
$C_5 + C_1.3$	$\{x/m, z/h\}$	$C_6 - A(m, y) - A(y, h)$
$C_6 + C_2$	$\{y/b\}$	$C_7 - A(b, h)$
$C_7 + C_1.3$	$\{x/b, z/h\}$	$C_8 - A(b, y) - A(y, h)$
$C_8.1 + C_3$	$\{y/t\}$	$C_9 - A(t, h)$
$C_9 + C_4$	\emptyset	$C_{10} \square$

La dernière Résolution a pour résultat la clause vide ; elle exhibe la contradiction et, dans ce système d'axiomes, le théorème $A(m, h)$ a donc été prouvé.

Prenons un second exemple, plus compliqué : « **Si dans un groupe le carré de tout élément est égal à l'élément neutre, alors le groupe est commutatif** ».

Nous avons besoin d'un prédicat pour traduire l'égalité du composé de deux éléments par la loi du groupe avec un troisième, soit | ce prédicat :

$$|(xyz) \text{ si et seulement si } x.y = z$$

Les propriétés liées à l'élément neutre e peuvent alors s'énoncer :

$$+ |(e, x, x) \text{ et } + |(x, e, x) \quad (3.8)$$

Il est bien entendu que les deux variables x ci-dessus sont indépendantes, comme quantifiées séparément ; nous les noterons dans la suite x_1 et x_2 pour qu'il n'y ait pas de confusion dans les unifications. Rappelons de plus qu'avant toute unification, donc avant de lancer toute Résolution, il faut toujours prendre soin de renommer différemment les variables qui étaient, à l'origine, quantifiées indépendamment dans les clauses : c'est la *séparation des variables*.

Si $|(y)$ désigne la fonction de Skolem qui donne l'inverse d'un élément y , il vient encore :

$$+ |(|(y), y, e) \text{ et } + |(y, |(y), e) \quad (3.9)$$

mais là aussi, les deux variables y sont indépendantes. Il conviendra de les séparer. Il nous faut maintenant exprimer l'associativité de la loi interne. Ceci se fait en deux temps ; traduisons d'abord l'associativité gauche :

$$x.(y.z) \mapsto (x.y).z \quad (\text{AG})$$

Ne disposant que du seul prédicat d'égalité I, il nous faut passer par des intermédiaires : soient $x.y = u$, $y.z = v$ et $x.v = w$; alors (AG) s'énonce :

« sous ces trois hypothèses, on a aussi $u.z = w$ »

expression dont la clause équivalente, sous forme normale disjonctive (puisque l'on a : $(A \wedge B \wedge C) \supset D \leftrightarrow (\neg A \vee \neg B \vee \neg C \vee D)$) n'est autre que :

$$- |(x, y, u) - |(y, z, v) - |(x, v, w) + |(u, z, w) \quad (\text{AG})$$

De façon semblable, se déduit l'associativité droite :

$$- |(x, y, u) - |(y, z, v) + |(u, z, w) - |(x, v, w) \quad (\text{AD})$$

La propriété donnée s'écrit quant à elle :

$$|(x, x, e)$$

Enfin, la négation de la conclusion :

$$(\forall x)(\forall y)(\forall z)((x.y = z) \supset (y.x = z))$$

puisque $\neg(A \supset B) \leftrightarrow \neg(\neg A \vee B) \leftrightarrow \neg\neg A \wedge \neg B \leftrightarrow A \wedge \neg B$, s'écrit :

$$(\exists a)(\exists b)(\exists c)(a.b = c) \wedge (b.a \neq c)$$

Les éléments a , b et c seront donc simplement considérés dans la suite comme trois constantes ; la démonstration doit finalement aboutir à l'impossibilité de leur existence. Cette négation est elle-même traduite sous forme de deux clauses :

$$+ |(a, b, c) \text{ et } - |(b, a, c)$$

Le squelette de la preuve par Résolutions successives, réduit à l'indispensable, prend l'aspect de la table 3.6.

C_1	$+ (e, x_1, x_1)$	
C_2	$+ (x_2, e, x_2)$	
C_3	$+ (l(y_1), y_1, e)$	
C_4	$+ (y_2, l(y_2), e)$	
C_5	$+ (x_3, x_3, e)$	
C_6	$- (x_4, y_4, u_4) - (y_4, z_4, v_4) - (x_4, v_4, w_4) - (u_4, z_4, w_4)$	
C_7	$- (x_5, y_5, u_5) - (y_5, z_5, v_5) - (u_5, z_5, w_5) - (x_5, v_5, w_5)$	
C_8	$+ (a, b, c)$	
C_9	$- (b, a, c)$	
<hr/>		
Clauses parentes	Substitutions	Résolvantes
$C_5 + C_{7.1}$	$\{x_5/x_3, y_5/x_3, u_5/e\}$	$C_{10} - (x_6, z_6, v_6) - (e, z_6, w_6) + (x_6, v_6, w_6)$
$C_8 + C_{10.1}$	$\{x_6/a, z_6/b, v_6/c\}$	$C_{11} - (e, b, w_7) + (a, c, w_7)$
$C_{11.1} + C_1$	$\{w_7/b\}$	$C_{12} + (a, c, b)$
$C_9 + C_{7.4}$	$\{x_5/b, v_5/a, w_5/c\}$	$C_{13} - (b, y_8, u_8) - (y_8, z_8, a) - (u_8, z_8, c)$
$C_5 + C_{13.1}$	$\{x_3/b, y_8/b, u_8/e\}$	$C_{14} - (b, z_9, a) - (e, z_9, c)$
$C_{14.2} + C_1$	$\{z_9/c\}$	$C_{15} - (b, c, a)$
$C_{15} + C_{6.1}$	$\{u_4, z_4/c, w_4/a\}$	$C_{16} - (x_{10}, y_{10}, b) - (y_{10}, c, v_{10}) + (x_{10}, v_{10}, a)$
$C_{12} + C_{16.1}$	$\{x_{10}/a, y_{10}/c\}$	$C_{17} - (c, c, v_{11}) - (a, v_{11}, a)$
$C_{17} + C_5$	$\{x_3/c, v_{11}/e\}$	$C_{18} - (a, e, a)$
$C_{18} + C_2$	$\{x_2/a\}$	$C_{19} \square$

TAB. 3.6 – Démonstration de la commutativité dans un groupe où : $\forall x x.x = e$, par la méthode d'Herbrand ou principe de Résolution

3.11.5 Analyse de ces démonstrations.

Bien que ces preuves ne nous soient pas très intuitives, on les reconstitue facilement en notation habituelle. La première établit successivement, à rebours, à partir de la conclusion :

$$\begin{aligned}
 (\forall y)m \in y &\supset y \notin h \\
 b \notin h &\quad \text{« un bras n'appartient pas à un homme, »} \\
 (\forall y)b \in y &\supset y \notin h \\
 t \notin h &\quad \text{d'où la contradiction.}
 \end{aligned}$$

Beaucoup d'autres Résolutions peuvent être effectuées et plusieurs chemins différents mènent à la clause vide. C'est en fait un handicap d'avoir un système de clauses avec un prédicat unique, car précisément, cela augmente le nombre de Résolutions possibles. Ainsi, dans la seconde démonstration, on voit que finalement, les deux clauses hypothèses sur les inverses $i(x)$ **ne servent pas** : toutes les étapes, et elles sont multiples, qui contenaient des Résolutions avec C_3 ou C_4 ont été ôtées. Un programme ne les comportant pas serait moins géné : il faut ici plus de cent étapes pour obtenir le tableau 1. Dans la preuve résumée dans ce tableau, la première étape importante est C_{12} , qui dit qu'avec les hypothèses on a : $ac = b$. Ce résultat sert de lemme, puisque la recherche repart ensuite dans une nouvelle direction pour obtenir C_{13} : $((by = u) \wedge (yz = a)) \supset (uz \neq c)$, clause partielle un peu étrange pour nous ; puis :

$$C_{16} : xy = b \supset x(yc) \neq a, \text{ et :} \tag{3.10}$$

$$C_{17} : a(cc) \neq a, \text{ d'où la contradiction puisque : } cc = e \tag{3.11}$$

Remarquons que la multiplication à gauche par c a été « intuitée » dans cette Résolution.

3.11.6 Stratégies

Un problème reste, bien sûr, entier devant l'explosion combinatoire des Résolutions : comment choisir convenablement les clauses et les littéraux à unifier pour atteindre le plus vite possible la clause vide ? Comme à l'accoutumée, il n'y a pas de réponse générale à cette question, **les choix doivent dépendre du contexte**. Les chercheurs ont cependant défini des stratégies de choix hors contexte, par limitation des Résolutions possibles. Il importe, ce faisant, de ne pas perdre la complétude, c'est-à-dire l'assurance d'atteindre encore, d'une façon au moins, toute preuve.

Stratégie unité {unit} : l'une des clauses parentes a toujours un seul littéral. Cette première stratégie perd évidemment la complétude, puisque, dès que le jeu de clauses ne comporte plus de clause à un seul littéral unifiable, c'est fini. Les démonstrations précédentes pouvaient être obtenues par cette stratégie.

Stratégie du donné {input} : l'une des clauses est une des hypothèses ou la négation de la conclusion. Cette stratégie n'est pas complète. La démonstration du premier exemple était dans ce cas.

Stratégie du support {set of support} : une des clauses parentes est la conclusion à démontrer ou bien l'une des clauses qui en est issue : **on part systématiquement du**

but et l'on essaye de remonter vers les hypothèses en s'interdisant toute Résolution à l'intérieur du système de départ. Cette stratégie est complète... mais longue, car elle introduit une structure de lemme qui n'est pas toujours très efficace.

Stratégie linéaire : une des clauses est la clause résolvante précédente et l'autre est une de ses ancêtres, ou bien une clause donnée. Cette stratégie, qui est une amélioration de la stratégie du donné, est complète : c'est **l'une des plus efficaces**.

D'autres stratégies ont encore été proposées dans la littérature. Remarquons que l'intersection de deux des stratégies précédentes donne une stratégie plus restrictive : celle-ci n'a, en particulier, aucune raison d'être complète même si ses origines le sont toutes deux.

Il reste encore à **choisir les littéraux** à l'intérieur des clauses parentes lorsque plusieurs unifications sont possibles. En général, les littéraux sont ordonnés et les programmes choisissent le premier couple qui donne un succès. Lorsque la numérotation des littéraux est transmise des parents à la résolvante, on obtient la stratégie dite de résolution *verrouillée* {lock-resolution}, qui est complète et efficace.

3.11.7 Intérêts et limites de la Résolution.

Cette méthode de démonstration est séduisante et puissante car systématique. De très nombreux articles en « démonstration automatique » lui ont été consacrés depuis les travaux d'Herbrand et la mise en application de Robinson en 1965. L'impossibilité de définir une stratégie valable dans tous les domaines entraîne un résultat bien concret : pour un jeu donné de clauses, ou bien la preuve est obtenue très vite sur ordinateur, ou bien elle ne sortira jamais. Les programmes étouffent en effet très vite dès que le nombre de clauses augmente. Le problème est alors de définir le seuil où l'on décide d'arrêter le programme puisque deux écueils existent : 1. Explosion du nombre de clauses ; 2. Arrêt prématuré alors que la preuve va être donnée...

L'importance et l'originalité de la Résolution tiennent en résumé à deux propriétés :

- a La gestion de l'arbre ET-OU, habituellement nécessaire à toute preuve, est éliminée. La pile standardisée des clauses joue ce rôle et une procédure très simple de traitement **choix + unification**, peut être définie ;
- b Les hypothèses, la conjecture et les clauses intermédiaires jouent exactement le même rôle. Les démonstrations sont obtenues en avançant à la fois dans les deux directions : hypothèses vers conclusions et conclusions vers hypothèses, la preuve est atteinte lorsque deux trajets parcourus se rencontrent (Figure 3.30).

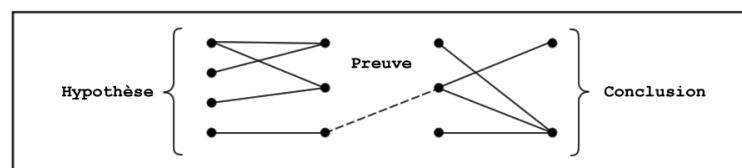


FIG. 3.30 – Schéma de preuve par Résolution

En revanche, un défaut majeur de la méthode est d'être, par construction, limitée à la **démonstration des seuls théorèmes**. Il n'est pas possible d'utiliser la Résolution pour inventer puis prouver des conjectures comme le fait le mathématicien. En outre,

si jamais la conjecture n'est pas un théorème, alors la Résolution peut développer un arbre infini... La Résolution est ainsi une excellente méthode simple et systématique, qui ne s'applique malheureusement qu'à un nombre restreint de cas : preuve peu profonde. Résolutions potentielles en faible quantité.

3.11.8 Le langage PROLOG et les applications de la méthode de Résolution

Dès 1972, Alain Colmerauer développait, avec son équipe à Marseille, un système bâti sur la logique du premier ordre et la méthode d'Herbrand-Robinson. L'utilisateur se sert du langage PROLOG pour poser son problème au système en termes de prédictats et celui-ci fournit automatiquement la réponse si elle existe par unifications et Résolutions. PROLOG peut donc être considéré comme un langage de programmation déclaratif où seule est donnée la définition du problème.

Comme l'utilisateur est ici intéressé à la solution efficace de son problème, il importe de la lui donner explicitement, la preuve par réfutation fournie par Résolution ne suffisant pas. Green avait suggéré, en 1969, d'introduire à cet effet un prédictat de réponse, lié à la conclusion :

$$\neg \text{conclusion}(x) \vee \text{sort}(x), \text{ soit} : \text{conclusion}(x) \wedge \text{sort}(x)$$

dans le jeu initial de clauses, pour que l'instanciation de la variable x fournit la solution explicite : finalement le système sort x instancié. Montrons comment fonctionne ainsi PROLOG. Supposons que l'on désire joindre une personne au téléphone. Nos données seront exprimées à l'aide de quatre prédictats T, V, A et $J.J(p, n)$: on peut joindre la personne p par le numéro de téléphone n , $A(x, l)$: la personne x se trouve au lieu l , $T(l, m)$: le lieu l a pour numéro de téléphone m . Supposons de plus que Pierre, que l'on veut joindre, soit en visite chez son ami Jean. $V(y, z)$ exprime que la personne y est en visite chez la personne z . On a donc au départ : $V(\text{Pierre}, \text{Jean})$.

Il nous faut encore exprimer les propriétés utiles des prédictats : si y visite z et si z est en e , alors y est en e (Clause C_2) ; si p est en l et si l a pour téléphone n alors on peut joindre p au numéro n (Clause C_1).

En PROLOG les signes \vee sont omis à l'intérieur des clauses et la négation est simplement marquée par le symbole \neg . PROLOG effectue automatiquement la séparation des variables lors des Résolutions successives. Toutes les clauses relatives à un même prédictat conclusion (précédé donc du signe $+$) doivent se suivre. En outre PROLOG n'admet que les clauses de Horn, c'est-à-dire celles qui comportent au plus un seul littéral positif qui doit être placé en tête. Le système de clauses suivant est ainsi fourni à PROLOG :

C_1	$:$	$+$	$J(x, m) - A(x, l) - T(l, m)$
C_2	$:$	$+$	$A(y, e) - V(y, z) - A(z, e)$
C_3	$:$	$+$	$A(\text{Jean}, \text{Bureau-Jean})$
C_4	$:$	$+$	$V(\text{Pierre}, \text{Jean})$
C_5	$:$	$+$	$T(\text{Bureau-Jean}, 46-54-32-10)$
C_6	$:$	$-$	$J(\text{Pierre}, n, \text{sort}(n) !)$

PROLOG fournit directement la réponse $n = 654 - 32 - 10$.

Les Résolutions suivantes sont transparentes à l'utilisateur, la stratégie est celle du support avec verrouillage : PROLOG développe son arbre de recherche en profondeur d'abord en partant de la clause but (seule clause négative). Les autres clauses sont prises dans l'ordre naturel. Les littéraux sont systématiquement étudiés de gauche à droite. Il utilise de préférence pour une nouvelle résolution, la dernière résolvante obtenue. Le backtrack est incorporé dans l'interpréteur lors de cette recherche.

$C_6 + C_{1.3}$	$\{x/\text{Pierre}, m/n\}$	$C_7 : -A(\text{Pierre}, l) - T(l, n) - \text{sort}(n)$
$C_{7.1} + C_2$	$\{y/\text{Pierre}, e/l\}$	$C_8 : -V(\text{Pierre}, z) - A(z, l) - T(l, n) - \text{sort}(n)$
$C_{8.1} + C_4$	$\{z/\text{Jean}\}$	$C_9 : -A(\text{Jean}, l) - T(l, n) - \text{sort}(n)$
$C_{9.1} + C_3$	$\{l/\text{Bureau Jean}\}$	$C_{10} : -T(\text{Bureau Jean}, n) - \text{sort}(n)$
$C_{10} + C_5$	$\{n/46-54-32-10\}$	$C_{11} : -\text{sort}(46 - 54 - 32 - 10)$

Le prédicat sort devenant ici unique littéral, la réponse est imprimée par le système.

La réponse donnée par PROLOG peut être non seulement une constante mais une variable formelle comme dans l'exemple : « Si x est le père de y et si z est le père de x alors z est le grand-père de y (C_1) et tout le monde a un père (C_2). Quel est le grand-père de t (t quelconque) ? »

L'énoncé :	$C_1 : +GP(z, y) - P(x, y) \quad P(z, x)$
	$C_2 : +P(p(u), u)$
	$C_3 : -GP(v, t) - \text{sort}(v)$

conduira aux Résolutions :

$C_3 + C_1 \quad \{z/v, y/t\}$	$C_4 : -P(x, t) - P(v, x) - \text{sort}(v)$
$C_4 + C_2 \quad \{x/p(u), u/t\}$	$C_5 : -P(v, p(t)) - P(v, x) - \text{sort}(v)$
$C_5 + C_2 \quad \{v/p(u), u/p(t)\}$	$C_6 : -\text{sort}(p(p(t)))$
sortie : $p(p(t))$	

On voit là la richesse du système et la possibilité d'applications à tous les univers où les arbres de démonstration ne sont pas trop gros. Un domaine important est **l'interrogation de base de données**. Lorsqu'on veut extraire d'une base les enregistrements qui possèdent des attributs désirés, le langage de la logique fournit toute la puissance voulue pour poser les questions. Un système du type PROLOG peut alors tout de suite aller chercher les réponses.

Une autre application concerne la **commande de robots**, il s'agit ici pour un robot de résoudre « de tête » un problème et d'en déduire un plan d'action. La même technique de mémorisation à l'aide du prédicat sort va permettre de retenir le plan échafaudé par Résolution. Le robot peut marcher, pousser un objet, grimper sur une caisse, il doit saisir des bananes accrochées au plafond et pour cela se servir de la caisse. Soit $E(x, y, z, s)$ le prédicat qui exprime : « Le robot en x , une caisse en z , les bananes en y définissent l'état s ». Soient :

- M(x₁, y₁, s₁) :** l'état atteint lorsque le robot **marche** de x₁ jusqu'à y₁ en partant de l'état s₁ ;
- P(x₂, y₂, s₂) :** l'état atteint lorsque le robot **pousse** la caisse de x₂ jusqu'à y₂ à partir de l'état s₂ ;
- G(s₃) :** l'état atteint si le robot **grimpe** sur la caisse à partir de l'état s₃ ;
- A(s₄) :** le prédicat qui exprime que le robot peut **atteindre** les bananes à partir de l'état s₄ ;
- a, b, c :* les positions initiales du robot, des bananes, de la caisse, (état s₀).

Clauses initiales :

		Le robot peut :
<i>C₁</i> :	+E(c, y, c, M(t, c, s)) - E(t, y, c, s)	- marcher vers la caisse
<i>C₂</i> :	+E(v, y, v, P(x, v, w)) - E(x, y, x, w)	- pousser la caisse
<i>C₃</i> :	+E(x, y, x, G(s)) - E(x, y, x, s)	- grimper sur la caisse
<i>C₄</i> :	+E(a, b, c, s ₀)	(état initial)
<i>C₅</i> :	+A(G(z)) - E(b, b, b, G(z))	- prendre les bananes
<i>C₄</i> :	-A(w) - sort(A(w))	(négation du but)

PROLOG résout sans problème cet exercice mais il lui faut une cinquantaine d'unifications. Allons ici au plus court en utilisant la stratégie suivante : partir du but, ne pas utiliser deux fois la même clause dans une branche de l'arbre, n'utiliser que la dernière résolvante et les clauses de départ.

Clauses parentes	Substitutions	Résolvantes
<i>C₅ + C₆</i>	{w/G(z)}	<i>C₇</i> : -E(b, b, b, G(z)) - sort(A(G(z)))
<i>C₇ + C₃</i>	{x/b, y/b, z/s}	<i>C₈</i> : -E(b, b, b, s) - sort(A(G(s)))
<i>C₈ + C₂</i>	{v/b, y/b, s/ P(x, b, w)}	<i>C₉</i> : -E(x, b, x, w) - sort(A(G(P(x, b, w))))
<i>C₉ + C₁</i>	{x/c, y/b, w/ M(t, c, s)}	<i>C₁₀</i> : -E(t, b, c, s) - sort(A(G(P(c, b, M(t, c, s)))))
<i>C₁₀ + C₄</i>	{t/a, s/s ₀ }	<i>C₁₁</i> : - sort(A(G(P(c, b, M(a, c, s ₀

A partir de la position initiale, le robot marche(M) de a vers la caisse en c qu'il pousse (P) alors en b sous les bananes, il ne lui reste qu'à grimper (G).

Chapitre 4

Les méthodes classiques en résolution de problèmes

Introduction

Il existe quatre grandes façons d'obtenir la solution d'un problème donné, c'est-à-dire d'exhiber un objet d'un ensemble connu satisfaisant des contraintes explicites imposées. Toute résolution procède ainsi soit par :

- A) Application d'une formule explicite donnant la solution ;
- B) Utilisation d'une définition récursive ;
- C) Utilisation d'un algorithme convergeant vers la solution ;
- D) Autre traitement, dont en particulier l'énumération des cas par « essai et erreur ».

La « meilleure » méthode, quand elle est utilisable, est sans aucun doute la première : on applique une formule, trouvée et prouvée auparavant, qui résout le problème posé dans tous les cas.

La détermination des zéros d'un polynôme du second degré, du nom du jour correspondant à une date quelconque, de l'intensité du courant dans un schéma électrique connu, sont des problèmes qui relèvent tous de cette approche directe. C'est d'ailleurs **l'objet même des mathématiques que d'obtenir**, pour le plus grand nombre de problèmes possibles, de telles **formules explicites**. Chaque fois qu'une formule explicite existe, la complexité du problème est liée au calcul effectif de cette formule. Celle-ci ne fait intervenir, par définition, qu'un nombre fini de symboles et d'opérations, et ceci quels que soient les paramètres intervenant dans cette formule, c'est-à-dire quelle que soit la taille n du problème d'entrée. La complexité est donc alors constante, indépendante de n , ou encore en $O(1)$. On considère ici la complexité de toute opération (soit : $+$, $-$, $/$, \times , $,$, etc) comme indépendante du nombre de chiffres des nombres sur lesquels elle porte. La complexité est alors simplement proportionnelle au nombre d'opérations.

4.1 Exemples de bons algorithmes

Exemple 1. « Calculer la somme des n premiers entiers »

Vous connaissez la formule explicite, de complexité constante : elle requiert une addition, une multiplication et une division.

$$\sum_{i=1}^n i = \frac{(n \times (n + 1))}{2}$$

Exemple 2. « Calculer la somme des carrés des n premiers entiers »

Peut-être savez-vous encore et pouvez-vous redémontrer que :

$$\sum_{i=1}^n i^2 = \frac{(n \times (n + 1) \times (2n + 1))}{6}$$

Exemple 3. « Calculer la somme des cubes des n premiers entiers »

Vous sentez venir la suite... **Mais ici l'informatique prend le relais des mathématiques.** Ne pas disposer de formule effective n'est pas grave, pour peu que l'on sache définir un processus fini de calcul permettant d'obtenir la solution. Une nouvelle question se pose au sujet de ce remplacement d'une formule par un processus : par combien d'étapes dois-je passer pour obtenir la solution ? C'est tout le propos de ce chapitre.

Souvent on dispose, dès l'énoncé d'un problème, d'une formule implicite de calcul où la solution est définie de façon récursive, de proche en proche, par rapport à elle-même. Ainsi, la somme $S(n)$ des n premiers entiers est définie par :

$$S(n) = S(n - 1) + n$$

et $S(1) = 1$

c'est-à-dire par le processus, directement accepté par un bon langage de programmation.

<pre> PROCEDURE S(n) : Somme des n premiers entiers SI n ≠ 1 ALORS S(n) ← S(n - 1) + n SINON S(n) ← 1 FSI FIN S </pre>

Pour calculer $S(5)$, un système informatique appellera la procédure S avec le paramètre formel n égal à 5 ; 5 est différent de 1, donc pour calculer le résultat qu'il doit donner, le système sait qu'il lui faudra ajouter 5 à la valeur $S(4)$, valeur qu'il lui faut donc calculer. Il appelle donc à nouveau la procédure S . Il mémorise en outre dans une « pile », le fait que quand il connaîtra $S(4)$, il devra additionner 5 pour obtenir ce qu'on lui demandait initialement. Cette information peut se représenter, pour nous, ainsi :

+	5	$S(4)$...
---	---	--------	-----

l'appel suivant développe le calcul de $S(4)$ et l'on a :

+	5	+	4	$S(3)$...
---	---	---	---	--------	-----

et ainsi de suite jusqu'à :

+	5	+	4	+	3	+	2	$S(1)$...
---	---	---	---	---	---	---	---	--------	-----

A cet appel $S(1)$ peut être enfin calculé par une formule explicite : $S(1) \leftarrow 1$.

Le système n'a plus besoin d'empiler ; il vide alors, à l'envers, la pile constituée pour trouver le résultat. Intuitivement, on voit que le processus précédent est rigoureusement équivalent au schéma itératif :

$S \leftarrow 0$	
REPETER	POUR $i \in [1 : n]$
	$S(n) \leftarrow S(n) + i$
FR SUR i	

Dans les deux cas, la complexité est en $O(n)$ puisqu'à chaque fois, c'est l'ordre de grandeur du nombre d'additions qu'il faut effectuer. Si maintenant on cherche la somme des carrés des n premiers entiers les deux derniers algorithmes sont aisément modifiés. D'une façon générale, la somme des puissances p^{imes} nécessite exactement $(n - 1)$ additions et n élévations à la puissance p .

Toute l'algorithmique consiste dès lors à construire des algorithmes de complexité acceptable pour tous les problèmes pour lesquels aucune formule directe effective n'est connue. En particulier, une méthode classique et bien agréable est de passer au départ par une écriture récursive qui est souvent rapide d'obtention, toujours concise et élégante. Dans le cas où la complexité de l'algorithme associé est trop élevée, le travail de l'informaticien sera ensuite de construire progressivement, à partir de la formulation récursive, une formulation plus directe, jusqu'à atteindre une complexité concrètement acceptable (« dérécursevation »).

Remarque :

La complexité théorique, dont il est ici question, n'est pas loin de la complexité bien concrète vécue par quelqu'un devant un terminal. Et les travaux des informaticiens théoriciens pour faire baisser la complexité des algorithmes pour certains problèmes courants reposent sur des constatations bien claires et bien réelles !

Exemple 4 :

Supposons que nous cherchions à connaître le 30^{me} nombre de Fibonacci. Ce nombre est défini par $F(30)$ avec les formules implicites :

$$\begin{cases} F(n) = F(n - 1) + F(n - 2) \\ F(2) = 1 \\ F(1) = 1 \end{cases}$$

Ces nombres, introduits par Léonard de Pise, fils de Bonacci (1540) interviennent dans plusieurs phénomènes naturels (reproduction de population par exemple : $F(n)$ est alors le nombre d'individus au temps n), et dans plusieurs branches des mathématiques (démonstration notamment de l'impossibilité du dixième théorème de Hilbert sur la résolution d'équations diophantiennes).

- A) **Formule explicite** : la résolution formelle et le calcul du terme général de la suite par recherche de combinaison linéaire de solutions de la forme r^n , où r est zéro du trinôme $x^2 - x - 1$, permet de donner l'expression exacte :

$$F(n) = \frac{[(1 + \sqrt{5})^n - (1 - \sqrt{5})^n]}{(2^n \times \sqrt{5})}$$

(formule de F. Binet)

Mais ce calcul ne nous est pas nécessairement connu. Par ailleurs, cette formule est ici relativement compliquée. Il suffit de calculer à la main $F(3)$ ou, a fortiori, $F(10)$ pour s'en convaincre : à la vérité le problème n'est pas même complètement résolu, puisqu'il faut encore spécifier l'algorithme de calcul de la formule. Doit-on ou non effectuer les développements des parenthèses internes par la formule du binôme ? Doit-on calculer formellement avec le symbole $\sqrt{5}$, ou numériquement avec la valeur 2,236068... et d décimales, et alors quelle valeur retenir pour d ? Le résultat doit être entier : il est clair cependant qu'avec un calcul en arithmétique finie, sur une machine de poche, ou même un gros ordinateur, le résultat ne sera pas entier : dès lors à quel entier « voisin » convient-il d'arrondir? Comment être absolument sûr du résultat? Voyons les autres approches : B et C.

- B) **Forme récursive** : dans ce problème, bien formulé et d'énoncé concis, la programmation récursive est la solution de remplacement immédiate :

Procédure $F(n)$: { Calcul du n^{ime} nombre de Fibonacci }
SI $n > 2$ ALORS $F(n) \leftarrow F(n - 1) + F(n - 2)$
SINON $F(n) \leftarrow 1$
FSI
FIN F

Mais à nouveau, comme pour la première solution, une difficulté se présente. Si-mulons, comme nous l'avions fait pour le calcul de la somme des premiers entiers, le déroulement de cet algorithme récursif par un système informatique. Voici le début de la construction de la pile pour le calcul de $F(30)$.

	+	$F(29)$	$F(28)$...				
	+	$F(29)$	+	$F(27)$	$F(26)$...		
	+	$F(29)$	+	$F(27)$	+	$F(25)$	$F(24)$...

Ceci, jusqu'à arriver à l'un des deux seuls résultats explicites connus :

	+	$F(4)$	$F(3)$...		
	+	$F(4)$	+	$F(2)$	$F(1)$...

Dans cette fin de pile, les bonnes substitutions sont enfin effectuées et le système remonte alors...jusqu'à $F(4)$ qu'il développe à nouveau, ... Comme, en tout état de cause, les seuls résultats connus, $F(2)$ et $F(1)$ valent 1, il fera finalement autant d'additions que $F(n)$ contient d'unités! Or $F(30)$ vaut 832040 : la pile de gestion des appels à la procédure F est du même ordre de grandeur. Ici, la méthode récursive brutale n'est tout simplement pas viable.

La place mémoire et le temps d'exécution requis ne correspondent à rien de réalisable. Il faut donc améliorer en faisant passer la complexité qui est ici en $O(F(n))$, soit quelque chose d'essentiellement exponentiel en fonction de n d'après la formule explicite, à quelque chose de « plus raisonnable ».

- C) **Algorithmes itératifs** : l'idée, pour ôter cette récursivité coûteuse, car très profonde, et éviter de faire et refaire toujours les mêmes calculs, est de transformer le schéma récursif précédent en schéma récurrent. Autrement dit, nous allons découper le problème en le supposant résolu... jusqu'à un certain point, c'est-à-dire une certaine valeur i de n . Ensuite nous en déduirons $F(j)$ pour au moins une

autre valeur j . Concrètement, ceci se fait en posant une **hypothèse de récurrence**. L'hypothèse naturelle ici est : $F(1), F(2), \dots, F(i)$ ont déjà été obtenus. Initialement, i vaut 2. Par ailleurs si $i = n$ le problème entier est résolu.

Sinon, la formule de définition $F(i + 1) = F(i) + F(i - 1)$ permet de faire porter l'hypothèse de récurrence un rang plus loin : $F(1), F(2), \dots, F(i), F(i + 1)$ sont alors connus. De plus, la formule de définition montre bien que les termes $F(1), F(2), \dots, F(i - 2)$ n'interviennent pas. La seule information qu'il est nécessaire de propager par récurrence est le triplet :

$$[i, F(i), F(i - 1)]$$

Si, par hypothèse, je suppose ce triplet connu, je sais en déduire le triplet suivant, c'est :

$$[i + 1, F(i + 1), F(i)] \equiv [i + 1, F(i) + F(i - 1), F(i)]$$

Et j'ai simplifié l'hypothèse de récurrence et mon schéma de calcul. J'ai ainsi l'algorithme itératif cherché.

Il apparaît que je n'avais pas besoin de retenir le vecteur $F[1 : n]$ dans son ensemble, mais que deux valeurs suffisent. Appelons ces valeurs u et v ; u correspond à $F(i)$ et v à $F(i - 1)$. Alors, du triplet :

$$[i, u, v]$$

je passe au triplet suivant qui est :

$$[i + 1, u + v, u]$$

$$\left[\begin{array}{ll} i \leftarrow i + 1 & \{ \text{nouvelle valeur de } i \} \\ w \leftarrow u & \{ \text{sauvegarde de l'ancienne valeur de } u \} \\ u \leftarrow u + v & \{ \text{nouvel } u \text{ par la formule de définition} \} \\ v \leftarrow w & \{ \text{nouveau } v = \text{ancien } u = F(i) \} \end{array} \right]$$

Les initialisations et le test d'arrêt ont déjà été indiqués, de sorte que l'algorithme complet dans cette troisième approche s'écrit :

Procédure Fibonacci (n) <i>i</i> $\leftarrow 2$; $u \leftarrow 1$; $v \leftarrow 1$; {Hypothèse de récurrence : $[i, F(i) = u \text{ et } F(i - 1) = v]$ connus } REPETER TANT QUE $i \neq n$ <i>i</i> $\leftarrow i + 1$; $w \leftarrow u$; $u \leftarrow u + v$; $v \leftarrow w$; FR sur <i>i</i> RESULTAT <i>u</i>

Fin Proc

Cet algorithme travaille visiblement en $O(n)$ au lieu de $O(F(n))$ auparavant. Le calcul de $F(n)$ correspond précisément à $n - 2$ itérations comportant chacune 1 test, 2 additions et 4 affectations. Nous sommes ainsi parvenus à une complexité concrètement acceptable et à un « bon » algorithme, c'est-à-dire polynômial, tout à fait utilisable.

Ce travail de conception d'algorithme n'est pas toujours aussi simple : les structures manipulées sont souvent plus complexes que des nombres ou de banales expressions arithmétiques. Le paragraphe suivant développe la démarche précédente sur quelques exemples plus compliqués, mais sur lesquels on parvient finalement à un

algorithme polynômial. Nous dresserons ensuite la liste complète des problèmes à ce jour, ainsi « bien résolus ». Il semble cependant que certains problèmes résistent et résisteront toujours : on ne peut pas, pour eux, trouver de bons algorithmes. Ce sera l'objet du paragraphe IV.4.

Exemple 5 : trier n nombres donnés à l'aide de comparaisons

Les nombres sont, par exemple, des numéros d'enregistrement ou des clés d'un fichier. On conçoit que l'opération de tri d'éléments saisis en vrac soit très courante en informatique. Etablissons d'abord la borne inférieure théorique de la complexité de ce problème. Bien sûr, aucune formule explicite au sens habituel n'est ici disponible, il faut définir un processus de rangement effectif, disons par ordre croissant, des n nombres donnés. Pour faire cela, il nous va falloir les comparer deux à deux d'une façon ou d'une autre. L'opération élémentaire fondamentale ici est le test de comparaison : « a est-il plus grand ou plus petit que b ? » Or, remarquons-le, si l'on doit trier n nombres, ceux-ci peuvent se présenter a priori à l'entrée suivant $n!$ manières différentes.

Par ailleurs, t tests successifs parviennent au mieux, dans un arbre équilibré de hauteur t , comme dans la figure 4.1, à distinguer 2^t situations, chacune correspondant à une feuille.

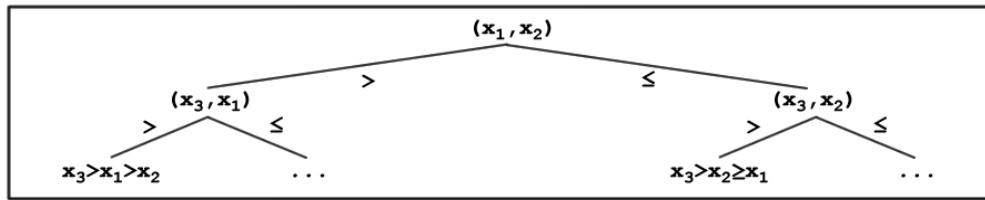


FIG. 4.1 – Arbre de test pour un tri

Il apparaît que t opérations élémentaires permettront au plus d'étudier n nombres, où n est tel que :

$$2^t \geq n! \quad (1)$$

En passant aux logarithmes et en utilisant la formule de Stirling, qui dit que $n!$ est équivalent à $n^{n+\frac{1}{2}}$ lorsque n tend vers l'infini, on trouve pour la borne inférieure de la complexité t d'un tri par comparaisons :

$$t = n \times \log_2 n \quad (2)$$

Il faut donc au moins de l'ordre de $O(n \times \log n)$ tests pour trier un ensemble de n éléments.

Examions maintenant le comportement d'un algorithme élémentaire de tri. L'idée la plus immédiate pour trier un tableau est de chercher le plus grand élément, de le placer en tête et de recommencer sur les autres, soit le schéma :

REPETER POUR $i = 1$ à $n - 1$ {les $i - 1$ premiers éléments sont triés}

$k \leftarrow i$

REPETER POUR $j = i + 1$ à n

| Si $x_k < x_j$ ALORS $k \leftarrow j$

FR sur j

échanger x_i et x_k

FR sur i

La boucle interne entraîne $(n - i)$ tests à chaque itération, soit au total $(n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n \times (n-1)}{2}$ tests pour n nombres absolument quelconques qui doivent être triés : cet algorithme possède une complexité en $O(n^2)$.

	1	2	3	4	5	6	7	8	
	20	7	84	75	45	78	30	9	
$k = 3$	84	7	20	75	45	78	30	9	
$k = 6$	84	78	20	75	45	7	30	9	
...	

FIG. 4.2 – Exemple de tri 1

Mais il est possible d'améliorer la méthode. Informellement ceci est dû au fait qu'on perd de l'information dans cet algorithme, en oubliant continuellement les résultats des comparaisons des tests de l'étape précédente. Une façon d'économiser ces comparaisons est de changer la boucle REPETER POUR j en un « tournoi » : à une étape on ne compare x_j qu'avec x_{j+1} , puis on compare le seul vainqueur au vainqueur de la paire qui jouxte celle-ci. Si l'entrée est 20, 7, 84, 75, 45, 78, 30, 9, il vient alors la figure 4.3 :

D'une part, nous réduisons ainsi le nombre de tests pour trouver le maximum. D'autre part, nous possédons des renseignements pour continuer la recherche. La hauteur de l'arbre est maintenant de l'ordre de : $\log_2 n$, toujours en vertu de l'inégalité (1). Le nombre d'opérations requis sera donc dans la pire des situations, n fois le nombre de tests nécessaires pour explorer un arbre de n éléments, soit : $n \times \log_2 n$. La complexité de ce tri, au moins dans son ordre de grandeur, est donc optimale d'après la relation (2).

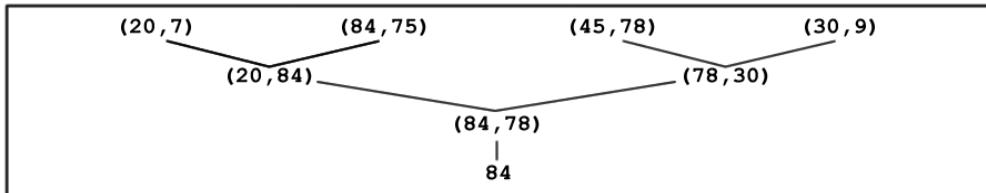


FIG. 4.3 – Exemple de tri 2

[En fait, la réalisation de ce tri sur ordinateur nécessite de préciser la façon dont est géré l'arbre où sont mémorisés les nombres et dont s'effectuent, dans cet arbre, les échanges entre les éléments triés et les autres. Tout ceci est important puisqu'initialement l'arbre n'est pas construit. On y parvient progressivement en passant par un intermédiaire, « le tas », structure où tous les descendants d'un noeud sont plus petits que lui.] En tout état de cause, nous avons ainsi un exemple de problème non trivial pour lequel on sait exhiber un algorithme polynômial optimal.

Exemple 6 :

Trouver le plus court chemin qui mène d'un point à un autre est un problème que nous rencontrons dans la vie de tous les jours au sujet de voyages par la route, de trajets en métro, ou encore pour résoudre certains casse-tête mathématiques. Soit par exemple le graphe de la figure 4.4, dans lequel il s'agit de se rendre du point 1 au point 6 par un chemin tel que la somme des distances, indiquées sur chaque liaison, soit minimale.

Les distances entre sommets sont aussi appelées, plus généralement, des *valuations*. Intuitivement, il semble qu'une énumération complète de tous les chemins venant de l'origine 1 à la sortie 6 soit inutile. En effet, il est tout d'abord clair qu'il convient de

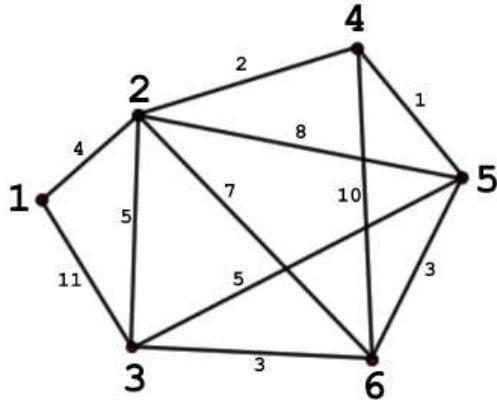


FIG. 4.4 – Un problème de plus court chemin

se limiter aux seuls trajets élémentaires, soit ceux qui n'empruntent qu'au plus une fois chaque arête du graphe, ceci du moins tant qu'on impose aux valuations de rester positives. Ensuite, l'idée naturelle est d'utiliser un processus itératif de propagation des distances minimales à partir du sommet 1. Calculant ainsi de proche en proche ces distances en commençant par les sommets voisins de 1, c'est-à-dire liés à 1 par une arête, il nous faut simplement faire en sorte qu'à un moment donné, pour un sommet donné, nous soyons sûrs que la distance minimale au sommet 1 est définitivement connue et donc qu'il n'y aura plus à revenir en arrière. Autrement dit, à chaque étape, l'ensemble des sommets sera partitionné en deux-ensembles S^* et S tels que :

- pour tout sommet de S^* , on connaît la distance minimale depuis 1 ;
- pour les sommets de S , on ne sait pas.

Initialement, l'ensemble S^* est réduit au sommet 1. Le passage d'un sommet de S dans S^* se fera lorsque l'on pourra démontrer qu'aucun chemin, non encore examiné, ne peut donner une distance plus courte que la meilleure parmi celles déjà connues : ainsi dans l'exemple, au premier pas, on peut, depuis 1, atteindre les sommets 2 et 3 ; comme le coût pour aller en 3 est déjà strictement plus élevé que celui pour aller en 2 la distance, 4, entre 1 et 2 est nécessairement la plus petite, quels que soient les chemins (inconnus) qui, depuis 3, permettent de revenir en 2 .

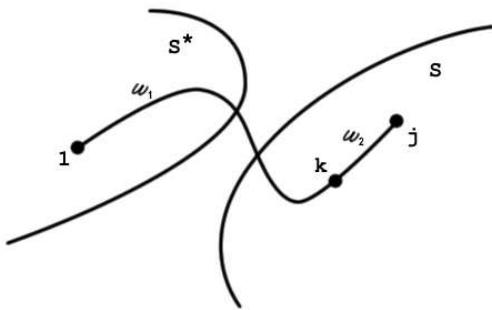


FIG. 4.5 – Preuve de l'algorithme : le chemin ω

Précisons et systématisons ce raisonnement au cas d'un graphe à valuations positives **quelconques**, afin de construire complètement l'algorithme cherché en notant :

- $D^*(i)$: la distance du sommet 1 au sommet i , $i \in S^*$;
 - $D(i)$: la meilleure distance connue du sommet 1 au sommet i , $i \in S$, à **une étape donnée** ;
- et

- $\ell(i, j)$ la valuation de l'arête (i, j) .

A toute étape, les ensembles S^* et S sont caractérisés par :

- Si $i \in S^*$ alors $D(i) = D^*(i)$

- Sinon, $i \in S$ et $D(i) = \text{MIN}_{k \in S^*, (k, i) \in U} [D(k) + \ell(k, i)]$

où l'ensemble U désigne l'ensemble des arêtes du graphe.

- $D(i)$ est bien alors la plus petite distance connue de 1 à i .

Initialement on a donc $S^* = \{1\}$ avec $D^*(1) = 0$ et $D(i) = +\infty$ pour tous les sommets autres que 1 qui sont donc tous dans S .

L'argument précédent pour les sommets 2 et 3 se généralise alors pour donner avec ces notations, la proposition :

Proposition : Soit $j \in S$ tel que $D(j)$ est la plus faible des $D(i)$ pour $i \in S$, alors $D(j)$ est la distance minimale de 1 à j .

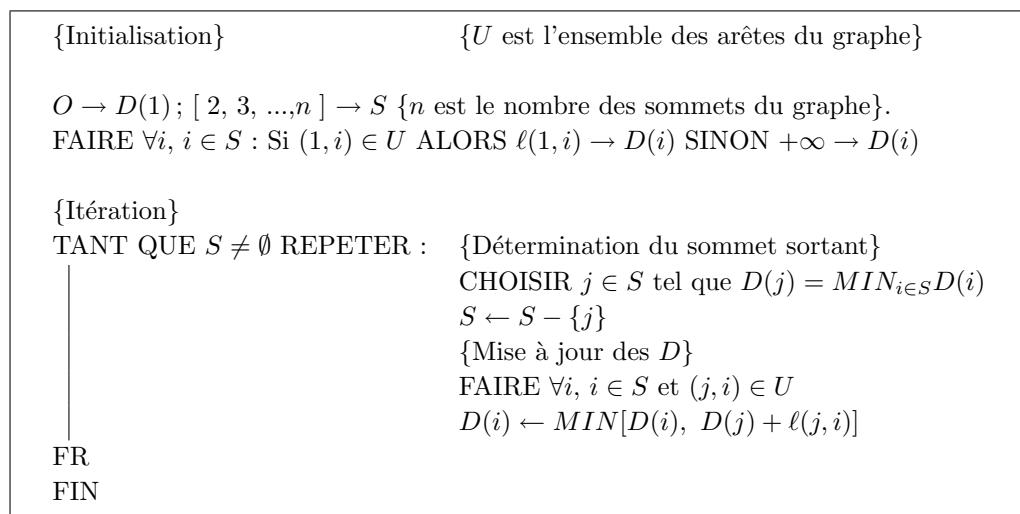
Ainsi, pour $j \in S$: $D(j) = \text{MIN}_{i \in S} [D(i)] \Rightarrow D(j) = D^*(j)$,

c'est-à-dire que, sous cette condition - qui est nécessairement satisfaite un j au moins à chaque étape -, on est en droit de faire passer j de S dans S^* .

Pour démontrer cette propriété, il nous faut prouver qu'il n'existe pas de chemin de 1 à j , plus court que celui que nous avons déjà et dont la distance $D(j)$ est enregistrée.

Considérons un chemin ω de 1 à j , et soit ω_1 la partie de ω allant de 1 au **premier** sommet, noté k , qui n'appartient plus à S^* , et ω_2 le complément (voir figure 4.5) de ω .

La longueur totale de ω est égale à celle de ω_1 augmentée de celle de ω_2 . Or, la longueur de ω_1 est supérieure ou égale, par construction, à $D(k)$ qui est, à ce stade, la distance minimale de 1 à k . Mais on a : $D(k) \geq D(j)$, d'après l'hypothèse de la proposition à démontrer. En outre, la longueur de ω_2 est positive ou nulle. Donc, au total, la longueur de ω ne peut être inférieure à $D(j)$. Le sommet j passe alors de S dans S^* , le processus s'arrête ainsi après $(n - 1)$ itérations, si n est le nombre de sommets, et finalement l'algorithme obtenu est le suivant :



Déroulons maintenant cet algorithme sur l'exemple de la figure 4.4.

- **Initialisation :**

	1	2	3	4	5	6
D	0	4	11	∞	∞	∞

- **Itération 1 :** $S = \{2, 3, 4, 5, 6\}$; sommet sortant $j = 2$, $MIN = 4$
 $(2, 3) \in U$ et $D(3) = MIN[11, 4 + 5] = 9$,
 $(2, 4) \in U$ et $D(4) = MIN[\infty, 4 + 2] = 6$,
 $(2, 5) \in U$ et $D(5) = MIN[\infty, 4 + 8] = 12$,
 $(2, 6) \in U$ et $D(6) = MIN[\infty, 4 + 7] = 11$.

- **Itération 2 :** $S = \{3, 4, 5, 6\}$ et $j = 4$, $MIN = 6$.
 $(4, 5) \in U$ et $D(5) = MIN[12, 6 + 1] = 7$,
 $(4, 6) \in U$ et $D(6) = MIN[11, 6 + 10] = 11$.

- **Itération 3 :**
 $(5, 3) \in U$ et $D(3) = MIN[9, 4 + 5] = 9$,
 $(5, 6) \in U$ et $D(6) = MIN[11, 7 + 3] = 10$.

- **Itération 4 :** $S = \{3, 6\}$ et $j = 3$, $MIN = 9$.
 $(3, 6) \in U$ et $D(3) = MIN[10, 9 + 3] = 10$.

- **Itération 5 :** $S = \{6\}$ et $j = 6$, puis FIN.

Au plus court chemin du sommet 1 au sommet 6 correspond donc une distance totale 10.

Nous remarquons que, dans cet algorithme, l'ordre dans lequel les sommets passent de S^* dans S dépend à la fois de l'existence des arêtes de U et de leurs valuations : sans changer le graphe, une modification des valuations change l'ordre. De plus, cet algorithme fournit en bloc les distances minimales de 1 à tous les autres sommets, si du moins on ne décide pas de l'arrêter dès que l'extrémité finale choisie, ici 6, passe dans S^* , éventuellement avant la fin. comme cela aurait pu se produire. Enfin, il nous est possible d'évaluer avec précision le nombre d'opérations élémentaires requis par cet algorithme : - fondamentalement, comme une arête dans un graphe relie, par définition, exactement deux sommets et que l'étape de mise à jour des $D(i)$ concerne le sommet qui vient d'entrer dans S^* et un autre de S , chaque arête intervient ici exactement une fois. Au total, au cours de l'ensemble des itérations, cette étape de mise à jour demandera donc, m additions et m comparaisons, si m est le nombre d'arêtes du graphe.

En outre, si l'ensemble S est représenté par son vecteur caractéristique, la détermination du sommet sortant demande $|S|$ comparaisons pour choisir le minimum des $D_n(j)$, soit, sur l'ensemble de l'algorithme :

$$\sum_{k=1}^n (n - k)$$

comparaisons, c'est-à-dire $\frac{n \times (n-1)}{2}$. L'algorithme proposé demande donc alors au total de l'ordre de $(n^2 + m)$ comparaisons et m additions. Considérant que la comparaison et l'addition sont des opérations qui sont exécutées en des temps comparables et que dans un graphe on a toujours, par définition, $m \leq n^2$, puisqu'une arête relie deux sommets, on dira finalement que le temps d'exécution de cet algorithme est de complexité $O(n^2)$.

{Cet algorithme a été proposé pour la première fois en 1957 par Moore puis, E. Dijkstra (1959), et des variantes sont dues à G. Dantzig (1960), Whiting et Hillier (1960)}.

Nous avons présenté un exemple dans lequel les liaisons appelées arêtes pouvaient être parcourues dans les deux sens, le contexte impose quelquefois un sens de parcours unique, on dit que le graphe est alors orienté : le sens de parcours sur chaque liaison est alors indiqué par une flèche et une liaison orientée s'appelle un arc. Tout ce que nous avons fait reste valable, puisque nulle part la notion d'orientation n'est intervenue ; il suffit de poser : $U = \text{ensemble des arcs}$.

Si (i, j) appartient à U , i est dit prédécesseur de j , et j successeur de i .

Exemple 7 : Un problème d'ordonnancement.

« Dans un atelier de montage automobile, le travail est décomposé en un certain nombre de tâches de durées connues. Ces tâches doivent se suivre dans le temps en respectant des relations d'antériorité logiques imposées par des contraintes physiques. Le tableau qui suit résume les informations dont on dispose.

Quelle est la durée minimale de montage, en supposant qu'on dispose d'une main-d'œuvre qualifiée suffisante pour pouvoir assurer les tâches en parallèle quand cela est possible ? »

Tâche	code de la tâche	durée en heures	relation d'antériorité
Emboutissage des tôles	A	1	-
Fixation du châssis	B	2	A.
Pose boîte et moteur	C	2	B.
Assemblages	D	3	B.
Pose de la direction	E	2	C.
Cablerie et réseau électrique	F	5	(), E.
Fixation des roues	G	1	C.
Pose des portes	H	1	B.
Peinture	I	2	D, H.
Pare brise et vitres	J	1	F, I.
Finitions et habitacle	K	4	F, I.
Vérifications et contrôle	L	2	J

Pour répondre à cette question associons à chaque tâche un point et traduisons les relations d'antériorité par des liaisons entre ces points : nous obtenons ainsi de façon naturelle un graphe, orienté cette fois, puisque « i précède j » est une relation antisymétrique traduite par un arc de i vers j . Portons de plus la durée de la tâche i sur cet arc comme une valuation.

En ajoutant deux tâches fictives, l'une, α , qui marque le début du montage et l'autre, ω , qui correspond à la fin du travail, nous dessinons le graphe de la figure 4.6.

Le tracé de ce graphe montre notamment, par construction, que ω est précédée des tâches G , K et L .

Ce dessin contient d'ailleurs, sous une forme condensée et accessible, toutes les informations nécessaires pour répondre au problème posé. Les contraintes d'antériorité comme C avant E , B avant C , A avant B , s'enchaînent mutuellement dans un ordre imposé, correspondent sur le dessin à des chemins : ici le chemin $\alpha ABCE$. Trouver la durée minimale de montage est donc équivalent à déterminer dans le graphe, valué par les durées, le chemin de valuation totale **maximale** entre les sommets α et ω . C'est bien le maximum qu'il convient de considérer ici, puisqu'alors les autres chemins ayant des durées au plus égales, ce maximum sera associé à la durée la plus petite possible pour réaliser l'ensemble des tâches.

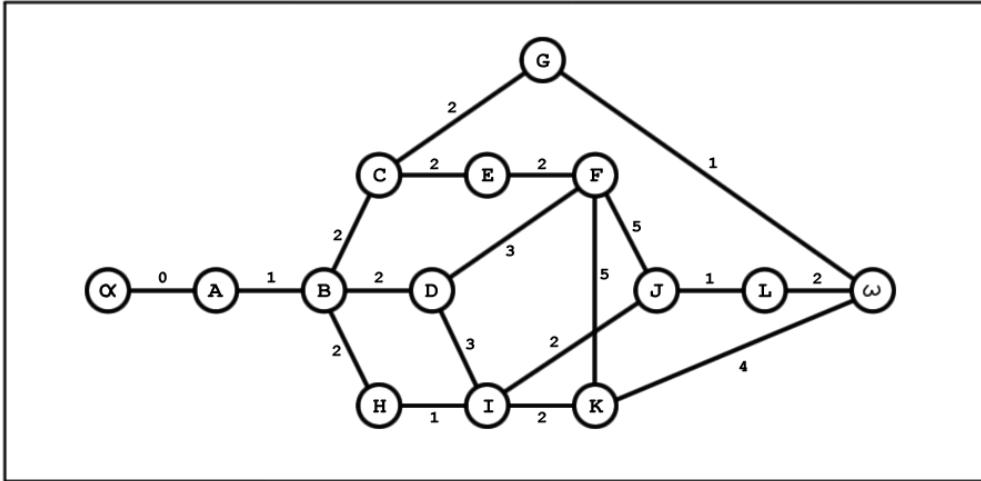


FIG. 4.6 – Graphe orienté pour un problème d'ordonnancement

Remarquons que, comme les valuations sont encore positives, mais que nous cherchons ici le chemin de valuation maximale, il pourrait se faire que cette dernière soit infinie. Cela se produirait en effet s'il existait un sous-ensemble d'arcs tel que :

$$(x_1, x_2), (x_2, x_3), \dots, (x_{p-1}, x_p) \text{ avec } x_p = x_1$$

c'est-à-dire un sous-ensemble de sommets sur lequel on pourrait boucler indéfiniment. Un tel sous-ensemble est appelé en théorie des graphes un *circuit*. Mais, par la nature même du problème d'ordonnancement traité, cette situation ne saurait se présenter : elle impliquerait en effet que la tâche $x_1 = x_p$ soit elle-même son propre antécédent : cette situation traduirait donc une anomalie dans le système de contraintes d'antériorité imposé. Le travail correspondant ne serait pas faisable.

L'algorithme mis au point dans le paragraphe précédent, qui concernait les chemins minimum, peut ici être amélioré en tenant compte de cette absence de circuit. Le fait que l'on cherche un maximum au lieu d'un minimum ne modifie pas fondamentalement la méthode. Nous nous bornons une nouvelle fois aux chemins élémentaires (c'est à dire empruntant chaque arc au plus une fois).

L'idée de base sera semblable : un poids $D(i)$ encore attaché à tout sommet sera progressivement ajusté par une relation de la forme :

$$D(i) \leftarrow \text{MAX}_{j \in J} [D(j) + \ell(j, i)]$$

jusqu'à stabilisation. Mais, ici, un ordre astucieusement déterminé des sommets va permettre une convergence en une seule étape.

L'idée est la suivante : dans le graphe de la figure 4.6, la longueur maximale de α à A est connue, celle de B l'est donc également. Connaissant celle de B , les longueurs maximales de α à C , D et H sont connues à leur tour. Celle de α à i l'est alors également et définitivement par comparaison de $D(H) + \ell(H, i)$ avec $\ell(D, i)$ et ainsi de suite. On peut ainsi considérer les sommets du graphe dans un ordre tel que, lorsqu'on arrive à un sommet i quelconque, tous les prédécesseurs de i aient déjà été examinés. L'algorithme peut effectivement démarrer en vertu de la propriété :

Propriété : Un graphe sans circuit possède au moins un sommet appelé source qui n'a pas de prédécesseur.

Supposons en effet qu'il n'en soit pas ainsi et marquons un sommet quelconque. Par hypothèse x_1 n'est pas une source et il existe au moins un arc (x_2, x_1) . Marquons alors le sommet x_2 et itérons ce marquage en partant maintenant de x_2 . Cette procédure définit ainsi une suite infinie de sommets : $(x_1, x_2, x_3, \dots, x_k, \dots)$. Mais, comme le graphe a un nombre fini de sommets, c'est que cette suite contient un circuit qui se répète, ce qui est absurde par hypothèse.

Cette propriété, qui nous permet de démarrer, va servir de façon itérative : il suffit en effet d'ôter de l'ensemble U des arcs, les arcs issus de la source précédente pour déterminer un nouveau sommet sans prédecesseurs, sommet dont la distance maximale à α va donc pouvoir être calculée directement. Nous avons en fait un théorème un peu plus fort.

Théorème : Un graphe $G = (X, U)$ est sans circuit si et seulement si il existe une bijection ν de l'ensemble X des sommets dans l'intervalle des entiers de 1 à n , où $n = |X|$, telle que : $\forall(x_1, x_2), (x_1, x_2) \in U : \nu(x_1) < \nu(x_2)$

Autrement dit, on peut numérotter les sommets (par la bijection ν) de telle sorte que tous les prédecesseurs d'un sommet quelconque aient des numéros inférieurs au sien.

D'abord il est évident que si G comporte un circuit (x_1, x_2, \dots, x_k) on ne peut construire un telle bijection, puisqu'alors on aurait par hypothèse :

$$\nu(x_1) < \nu(x_2) < \dots < \nu(x_k) < \nu(x_1)$$

ce qui est impossible.

Réciproquement, si G est sans circuit, il existe une source que nous numérotions 1 : $\nu(S) = 1$. Otions alors ce sommet de G et tous les arcs qui en partent, le nouveau graphe, obtenu par soustractions est évidemment encore sans circuit : il possède donc une source que nous numérotions 2 et ainsi de suite.

S'il existait à un moment quelconque un arc (x_1, x_2) de U avec $\nu(x_1) < \nu(x_2)$, où x_2 est une source, alors que x_1 appartient encore au graphe restant, il y aurait une contradiction. La fonction ν existe donc bien et nous avons un procédé effectif pour la fabriquer.

Algorithme de construction de la fonction de classement des sommets

Dans cet algorithme chaque arc de U est considéré exactement une fois si bien que la complexité est en $O(m)$.

Il nous est maintenant possible de réagencer suivant cette procédure le graphe d'ordonnancement de l'exemple. Dans la figure 4.7 les ensembles Y_k , encore appelés *couches*, ont été visualisés.

Initialisation :

$$1 \rightarrow p; X \rightarrow T; 0 \rightarrow k$$

Construction :

TANT QUE $T \neq 0$ REPETER : $\{Y_k = \text{ensembles des sommets sans prédecesseurs}\}$

FAIRE $\forall i, i \in Y_k;$

$$p \rightarrow \nu(i); p + 1 \rightarrow p$$

FAIRE $\forall j, j \in X, (i, j) \in U :$

Oter (i, j) de $U.$

$$T - Y_k \rightarrow T; k + 1 \rightarrow k$$

FR

FIN

La première couche est évidemment α , la seconde A , la troisième B ; les arcs BC , BD , BH sont ôtés et C , D et H appartiennent à la couche Y_4 . CG et CE , DF et DI , enfin HI sont ôtés. Y_6 est constitué du seul sommet F ; FJ et FK sont ôtés; Y_7 contient donc J et K . Enfin, L et ω forment les dernières couches Y_8 et Y_9 .

Remarque : La numérotation ν n'est pas indispensable quand on a le dessin sous les yeux, aussi l'avons nous omise dans la figure 4.7.

Il vient alors :

$$D^*(1) = 0; D^*(2) = 0$$

Puis :

$j = 3,$	$D^*(3) = 0 + 1 = 1$
$j = 4,$	$D^*(4) = 1 + 2 = 3$
$j = 5,$	$D^*(5) = 1 + 2 = 3$
$j = 6,$	$D^*(6) = 1 + 2 = 3$
$j = 7,$	$D^*(7) = MAX[3 + 1; 3 + 3] = 6$
$j = 8,$	$D^*(8) = 3 + 2 = 5$
$j = 9,$	$D^*(9) = 3 + 2 = 5$
$j = 10,$	$D^*(10) = MAX[3 + 3; 5 + 2] = 7$
$j = 11,$	$D^*(11) = MAX[6 + 2; 7 + 5] = 12$
$j = 12,$	$D^*(12) = MAX[6 + 2; 7 + 5] = 12$
$j = 13,$	$D^*(13) = 12 + 1 = 13.$
$j = 14,$	$D^*(14) = MAX[5 + 1; 12 + 4; 13 + 2] = 16$

La longueur du plus long chemin dans le graphe du problème d'ordonnancement 6, est ainsi de 16 unités. Le chemin lui-même est retrouvé en notant à chaque itération le sommet prédecesseur qui donne effectivement le maximum lors du calcul de D^* . En reprenant ces informations à reculons à partir de ω , on obtient finalement le chemin :

$$\alpha A B C E F K \omega$$

La méthode que nous avons utilisée est connue sous le nom de « **méthode des potentiels** »; elle a été mise au point par Bernard Roy en 1960. Une variante américaine plus connue, mais aussi plus lourde, pour résoudre le même problème, a pour nom : « **méthode PERT** » (Program Evaluation and Review Technique), elle fut développée à la NASA et à la RAND (1954).

Ces méthodes sont journallement utilisées pour résoudre efficacement des problèmes concrets principalement dans l'industrie du bâtiment mais aussi dans des constructions de tout type : automobile, navale ou aérienne.

Outre la durée minimale d'exécution d'un projet, elles fournissent la liste des tâches dites « critiques », c'est-à-dire des tâches appartenant, dans la représentation en graphe, au plus long chemin déterminé.

Par définition, tout allongement de la durée d'une quelconque de ces tâches aura nécessairement une répercussion sur le délai de livraison. Les signaler au responsable des travaux donne une précieuse information. Un autre sous-produit utile de notre algorithme est la charge de travail des différents corps de métiers au cours du temps.

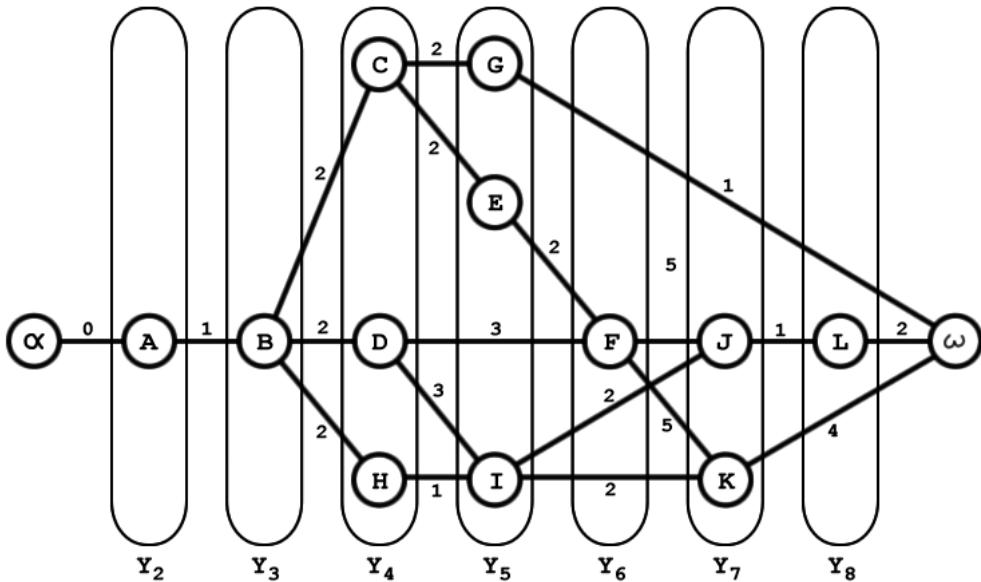


FIG. 4.7 – Graphe réarrangé pour le problème d'ordonnancement

Dans un graphe ainsi réarrangé, on dispose d'un algorithme trivial pour déterminer le chemin le plus long, c'est le suivant :

{Initialisation} $D^*(1) \leftarrow 0 ; D^*(2) \leftarrow \ell(1, 2)$.

REPETER POUR j $3 \leq j \leq n$ $D^*(j) \leftarrow \text{MAX}_{(i,j) \in U} [D^*(i) + \ell(i, j)]$ FR

En effet, la nouvelle numérotation est telle que :

$$(i, j) \in U \Rightarrow i < j$$

Ainsi, tous les chemins qui parviennent en j ne peuvent passer que par des sommets de numéros plus petits et dont le D^* est donc déjà calculé. Ceci montre, par récurrence sur j , la validité de cet algorithme.

L'itération requiert, pour chaque valeur particulière, $(j - 1)$ additions et $(j - 2)$ comparaisons, de telle façon que cet algorithme compte finalement de l'ordre de $\frac{n \times (n-1)}{2}$, soit $O(n^2)$ opérations élémentaires.

Sur le graphe de l'exemple (figure 4.6), la numérotation $\nu(x)$ est donnée par le tableau :

nom du sommet x	α	A	B	C	D	E	F	G	H	I	J	K	L	ω
$\nu(x)$	1	2	3	4	5	8	10	9	4	7	12	11	13	14

Enfin, parce que les données dans ce genre de problèmes ne peuvent jamais être connues avec exactitude, des techniques complémentaires de calculs ont été conçues pour suivre l'évolution du chemin critique et de la durée totale lorsque les durées des tâches sont paramétrées et varient dans des intervalles donnés avec des lois de probabilité connues.

Le développement de tous ces compléments n'est pas dans les buts de ce chapitre où nous avons voulu essentiellement donner, sur des exemples l'idée de la démarche algorithmique : analyse du problème pour le traduire sous une forme concise et agréable (un graphe) ; étude du nouveau problème dans cet univers réduit et ébauche d'une méthode guidée par des idées simples et naturelles sur un exemple ; généralisation puis

mise en forme et preuve de l'algorithme.

Exemple 8 : Un problème de topologie

Considérons maintenant le célèbre problème du « cycle eulérien » dans un graphe. Il s'agit de savoir s'il existe dans un graphe donné, tel celui de la figure 4.8 un cycle qui passe une fois exactement par toutes les arêtes et qui revient au point de départ.

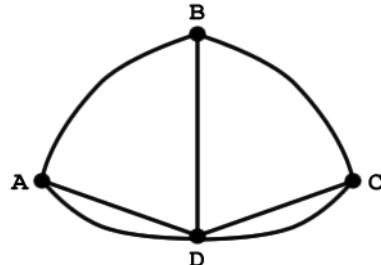
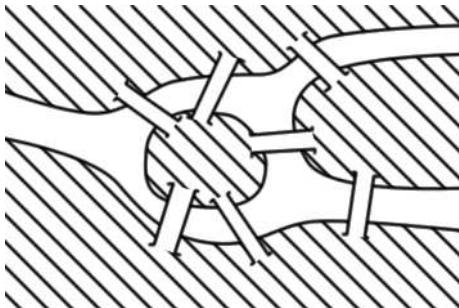


FIG. 4.8 – Les 7 ponts de la Pregel. Le problème du parcours eulérien

Une condition nécessaire est, bien sûr, que le graphe donné soit connexe c'est-à-dire qu'il existe déjà une chaîne d'arêtes permettant de passer de tout point à un autre. Une autre condition nécessaire concerne le nombre d'arêtes en chaque sommet : puisqu'on ne peut réemprunter une arête, il faut qu'à toute entrée soit associée une sortie, autrement dit.

« Pour que dans un graphe connexe il existe un circuit eulérien il faut que ce graphe ne possède pas de sommet de degré impair » (degré = nombre d'arêtes issues d'un sommet).

Cette condition est en fait suffisante, nous allons le montrer. Elle donne donc directement un algorithme polynômial pour répondre à la question de l'existence d'un circuit eulérien et va, en outre, fournir un algorithme, de complexité égale à l'ordre du nombre d'arêtes en entrée, pour la construction effective du circuit quand il existe.

Condition nécessaire : Elle est immédiate. Puisqu'une chaîne qui passe par toutes les arêtes parvient en chaque sommet et le quitte pour revenir finalement à son sommet de départ, il faut que tous les sommets traversés aient un nombre pair d'arêtes.

Condition suffisante : Elle est un peu moins immédiate.

Nous raisonnons par induction sur le nombre m des arêtes du graphe.

Constituons progressivement, sous les hypothèses du théorème, une chaîne Φ partant d'un sommet arbitraire s .

Pour être certains de ne pas réemprunter la même arête, convenons d'ôter du graphe toute arête parcourue. Tant que Φ aboutit à un sommet x autre que s . Il est toujours possible de la prolonger d'une arête au moins. En effet, tout sommet possède au départ un nombre pair d'arêtes et il en subsiste nécessairement encore un nombre impair (c'est-à-dire au moins une) au sommet courant x , puisque chaque passage en ce point en ôte exactement deux.

Si l'on ne peut prolonger la chaîne Φ , c'est donc qu'on est revenu en s .

Si, à ce moment, toutes les arêtes du graphe initial ont été parcourues, le théorème est démontré.

Sinon, il reste un certain sous-ensemble d'arêtes qui se répartissent en un certain nombre de composantes connexes (les composantes réduites à un sommet isolé sont,

bien sûr, ignorées ici).

Appelons ces composantes connexes G_1, G_2, \dots, G_k . Comme G est supposé connexe au départ, la chaîne Φ rencontre chacune de ces composantes en un point au moins. Soient :

g_1, g_2, \dots, g_k , un de ces points dans chacune des composantes G_i .

Puisque, quel que soit i , le sous-graphe G_i est connexe et possède, par construction, moins de m arêtes et des sommets de degrés pairs, il admet, par hypothèse d'induction, un cycle eulérien E_i .

Dès lors, on construit finalement et dans cet ordre, une nouvelle chaîne en empruntant :

- la chaîne Φ du sommet s au sommet g_1 ,
- le cycle eulérien E_1 dans G_1 depuis g_1 jusqu'à g_1 ,
- à nouveau Φ de g_1 jusqu'à g_2 et ainsi de suite,

tout en étant assuré de revenir en s , depuis g_k , par la chaîne Φ , avec $s \in G_k$.

Enfin, le fait qu'une boucle, sur un sommet unique, soit, à elle seule, un circuit eulérien achève cette démonstration, trouvée tout d'abord par Léonard Euler en 1736.

Cette condition nécessaire et suffisante est très simple à tester en chaque sommet avec une complexité totale en $O(n)$. Pour obtenir en outre la construction effective d'un cycle eulérien, écrivons d'abord une procédure Euler(w) qui fabrique, pour un graphe quelconque, un début de cycle eulérien commençant au sommet w .

<pre> Procédure Euler (w) $u \leftarrow w$; $v \leftarrow \Phi$ REPETER TANT QUE $v \neq w$: {Soit (u, v) la première arête, non marquée, issue de u } Marquer (u, v) $\Phi \leftarrow \Phi \cup \{(u, v)\}$ {augmenter la chaîne Φ } $u \leftarrow v$ FR FIN Euler </pre>
--

L'algorithme complet d'obtention d'un cycle eulérien s'écrit maintenant facilement. On construit une chaîne qu'on complète progressivement, en suivant la preuve du théorème.

<pre> $\Phi \leftarrow \text{Euler}(S)$ {C'est la chaîne initiale} REPETER TANT QU'il existe des arêtes non marquées dans G ; {Soit h le sommet, situé à la fois sur la chaîne Φ et sur les arêtes non marquées.} Concaténer la chaîne donnée par Euler(h) à la chaîne Φ FR </pre>
--

Dans cet algorithme, la procédure Euler(w) traite au plus une fois chaque arête de G . On se dispense donc de la construction effective des composantes connexes, en choisissant à la place un sommet convenable h comme indiqué.

Chaque arête est alors examinée deux fois dans le pire des cas. La complexité de l'algorithme global est ainsi de l'ordre du nombre d'arêtes du graphe initial, soit $O(m)$.

Nous possédons donc un excellent algorithme linéaire pour le problème d'Euler.

Remarque :

Nous renconterons tout à l'heure un problème a priori très voisin de ce problème d'Euler : c'est le problème, étudié notamment par Hamilton, qui lui a ainsi laissé son nom, de la recherche dans un graphe quelconque d'un cycle qui passe, non plus par toutes les arêtes, mais par tous les sommets (cf. chapitre 5 : problème du voyageur de commerce).

Eh bien, de façon très étrange, il apparaît que ce nouveau problème n'est pas du tout dans la même classe que le précédent. Après des siècles d'efforts, de nombreux mathématiciens, souvent de renom, n'ont pu construire, pour le problème d'Hamilton, un bon algorithme (c'est-à-dire polynômial), comme pour le problème d'Euler.

En d'autres termes nous ne savons pas résoudre le problème général de la détermination d'un cycle qui passe par tous les sommets d'un graphe.

4.2 Liste des problèmes bien résolus (algorithmes polynômiaux)

Nous avons d'abord vu qu'à chaque fois qu'on connaissait pour un problème une formule mathématique explicite, la complexité du problème était du même coup constante (indépendante des données) et que, bien sûr, c'était là le cas idéal.

Ce cas concerne par exemple la résolution des systèmes linéaires non dégénérés, le calcul d'une paye et la plupart des problèmes de physique rencontrés classiquement au lycée qui se ramènent à un système d'équations.

Nous complétons ici la liste par des problèmes plus précis faisant intervenir d'autres types de données.

Rappelons d'abord les deux derniers problèmes rencontrés :

- **Trier un ensemble de n nombres** : algorithme en $O(n \times \log n)$
- **Trouver un circuit eulérien dans un graphe de m arêtes** $O(m)$

D'autres résultats ont été obtenus pour d'autres types de problèmes. Nous les citons sans démonstrations.

- **Rechercher un mot dans un texte de longueur n** $n \times O(n)$

Il s'agit ici de trouver par exemple dans un fonds documentaire toutes les occurrences d'un (ou de plusieurs) mot(s) donné(s) : descripteurs ou mots-clés pour des références bibliographiques). L'algorithme évident, comportant deux boucles imbriquées sur les lettres du texte, puis sur celles du mot, est inapplicable concrètement car de complexité trop grande $O(n \times \text{nombre de lettres du mot})$. Il a été amélioré jusqu'à obtenir une complexité indépendante de la longueur du mot recherché : chaque caractère du texte n'est - en tous cas - analysé qu'une fois. Ceci est réalisé comme souvent en « échangeant du temps contre de l'espace » : la mise en mémoire du mot sous forme de « diagramme de transition » évite les tests inutiles.

- **Construire l'arbre recouvrant de coût minimum** ($m = \text{nombre d'arêtes}$) $O(m \times \log m)$ Kruskal (1956), Prim (1957), Tarjan (1977)

Il s'agit ici, par exemple, de construire le réseau d'alimentation en eau dans une ville de province. Plusieurs jonctions sont possibles, à chacune correspond son coût de construction, chaque sommet (client) doit être ravitaillé. En particulier, la source connectée au réseau le mettra tout entier sous pression, et, si les canalisations passent

partout (« recouvrant »), ne forment pas de circuits (« arbre »), tout un chacun sera ravitaillé. Il s'agit de construire un tel arbre pour un coût total minimum de construction des jonctions. Il se trouve qu'ici l'algorithme trivial : « choisir l'arête de plus petit coût qui ne forme pas de circuit avec celles déjà retenues et recommencer jusqu'à ce que tout sommet soit atteint » : 1) - donne bien l'arbre optimal ; 2) - est de complexité minimale.

De nombreux autres problèmes de graphes ont aussi été résolus de façon polynômiale. Donnons les principaux :

- **Plus court chemin dans un graphe de n sommets, m arcs** $O(m \times n)$

Ford (1965), Dijkstra (1959), Dantzig (1960), Floyd (1962)

Un graphe valué étant donné, on cherche le chemin reliant deux points donnés dont la somme des valuations des arêtes est minimale. Le problème complémentaire du chemin le plus long est de même complexité.

- **Composantes connexes :** $O(n^2)$

Trémaux (1882), Tarjan (1972)

Il s'agit de déterminer les sous-ensembles (composantes connexes) de sommets tels que deux sommets d'une même composante soient toujours reliés par une chaîne d'arcs.

On détermine ainsi, par exemple, dans un projet, les sous-ensembles de tâches complètement indépendants entre eux.

- **Fermeture transitive :** $O(n^2)$

Nolin (1964), Warshal (1965), Roy (1965)

On veut trouver pour tout sommet l'ensemble des sommets auquel il est relié directement ou indirectement. L'algorithme de recherche est semblable au précédent.

- **Couplage maximum :** $O(n^{\frac{5}{2}})$

Berge (1962), Edmonds (1965), Gabow (1973), Bartnik (1978)

Il faut trouver dans un graphe quelconque le plus grand sous ensemble possible d'arêtes qui ne se rencontrent en aucun sommet. Un théorème de Claude Berge donnait une CNS pour qu'un ensemble soit maximum, mais, c'est encore plus récemment, à Jack Edmonds que revient le mérite d'avoir montré que cette condition pouvait être testée en un temps polynômial. Il donnait en outre une amélioration effective de tout ensemble non maximum ; c'est dans un célèbre article « Chemins, arbres et fleurs » qu'il a inventé le terme « bon algorithme » et montra son importance.

Ce problème de couplage est sous-jacent à de nombreux problèmes de type affectation ou ordonnancement disjonctif (tâches s'excluant mutuellement dans un même intervalle de temps), organisation de rotation d'équipages, de moyens de transports, d'emplois du temps. Des problèmes un peu plus complexes en dérivent de plus immédiatement.

- **Flot maximum :** $O(n^3)$

Ford et Fulkerson (1950), Gondran et Minoux (1978)

Il s'agit de faire passer entre deux points une quantité maximum de « flot » sachant que chaque arc possède une capacité maximale qu'on ne peut dépasser. Les chargements de bateaux entre les ports, quelquefois ceux des trains ou des avions utilisent ce modèle pour optimiser les transports.

Certains problèmes de graphe ont vu, au fil des années, par le travail insistant des chercheurs, leur complexité diminuer de façon spectaculaire.

Donnons un seul exemple :

– **Test de planarité d'un graphe :** $O(n)$

Un graphe est planaire si et seulement si on peut le dessiner sur une surface plane de telle manière que ses arêtes ne se rencontrent jamais en dehors des sommets.

En 1930, Kuratowski donnait une méthode en $O(n^6)$ pour opérer ce test ; jusqu'en 1970, plusieurs polynômes de degrés s'abaissant finalement jusqu'à 2 furent obtenus. En 1970, Hopcroft et Tarjan exhibèrent une méthode en $O(n \times \log n)$ et finalement, en 1974, ils montrèrent qu'on l'améliorait pour parvenir à une complexité $O(n)$ et, du même coup, qu'on ne pouvait faire mieux.

Un cas plus étrange encore, est celui de problèmes pour lesquels on n'a pas connu, pendant très longtemps, d'algorithme polynominal, sans savoir montrer pour autant ni qu'il n'en existait pas, ni que le problème était nécessairement difficile car équivalent à des problèmes répertoriés comme tels.

Ceci a été très longtemps le statut de la programmation linéaire.

– **Programmation linéaire :**

Il s'agit de la résolution dans \mathbb{R}^n d'un système linéaire

$$AX = b$$

où A est une matrice m lignes, n colonnes, de coefficients donnés,

X est le vecteur des n inconnues réelles (positives), $X \in \mathbb{R}^n$, b est le vecteur des m seconds membres donnés.

L'ensemble des solutions X est éventuellement départagé, en gardant celui qui minimise en plus une fonctionnelle elle-même linéaire, de la forme cX (c vecteur de n coefficients donnés).

Cette formulation permet de modéliser de très nombreux, très gros (en taille), très importants (en gains monétaires attendus) problèmes dans des domaines variés : gestion de production, acheminement de frêt, optimisation de la distillation du pétrole, fabrication et distribution du courant électrique.

Elle est à la base du succès de la Recherche Opérationnelle depuis vingt-cinq ans, ainsi que du succès du créateur, - G. Dantzig- de la méthode utilisée aujourd'hui pour résoudre ce type de problème.

Mais l'algorithme de G. Dantzig, appelé *algorithme du simplexe*, est fondamentalement basé sur une heuristique de gradient (c'est à dire d'optimum local) ; s'il est assuré, par la convexité de l'espace de recherche, de parvenir à l'optimum global cherché. Il peut se faire qu'il y arrive seulement après être passé par un nombre exponentiel (en fonction de m et n) de points, c'est-à-dire d'itérations.

Or, depuis une vingtaine d'années, l'algorithme du simplexe, progressivement amélioré et complété, a été utilisé, à la vérité avec grande efficacité, un peu partout dans le monde. Concrètement, alors que la complexité théorique est en $O(n^m)$, on a constaté, même sur des problèmes de grosses dimensions ($m = 1000$, $n = 40000$ par exemple), une complexité expérimentale en $O(n^3)$!

Et puis, un jour, en 1979, un article russe de L. G. Khachian paraît à Moscou. Il est lu, vérifié, complété seulement un an plus tard en Europe et aux Etats-Unis, il exhibe un algorithme polynomial pour le problème de la programmation linéaire. Cet algorithme est très différent du simplexe et, s'il explique peut-être le succès expérimental de son prédecesseur, il n'a par ailleurs aucune chance de le détrôner tant, pour les problèmes concrets rencontrés, l'algorithme du simplexe donne satisfaction.

L'algorithme de Khachian, qui fit en 1979 la première page de plusieurs quotidiens

américains et canadiens, établit surtout finalement la convergence, en un nombre polynômial d'étapes et pour une précision donnée d'une certaine famille d'algorithmes en analyse numérique.

Ce qui est absolument remarquable c'est que cette récente découverte de Khachian clôt la liste des problèmes polynômiaux, c'est-à-dire des problèmes bien résolus : ils sont réellement très peu nombreux ! Bien sûr, il existe des sous-cas, des variantes, des formulations différentes par rapport aux cas cités. Bien sûr, quelques problèmes sont dans le cas de la programmation linéaire en 1978 : pas d'algorithme polynômial, pas d'équivalence démontrée avec d'autres problèmes répertoriés exponentiels. Il en est ainsi du problème de l'isomorphie de deux graphes quelconques.

Mais, il est clair que la classe des problèmes non résolus en temps polynômial est, en comparaison, absolument immense.

Mise à part la classe E des problèmes pour lesquels le temps de calcul doit a priori être exponentiel, le cas des problèmes pour lesquels rien ne s'oppose théoriquement à une complexité polynômiale, sans que pour autant aucun bon algorithme ne leur soit applicable se présente comme « l'énigmatique cas général » !

4.3 Le classement des problèmes selon leur complexité

Selon notre expérience, notre familiarité plus ou moins importante avec tel ou tel domaine, nous avons tous, individuellement, l'« intuition » que tel problème est « plus difficile » que tel autre.

Cette notion, strictement personnelle, n'est pas satisfaisante ; mais peut-on définir la complexité intrinsèque d'un problème ?

C'est à cette définition et au classement des problèmes qui en découle, que va s'intéresser ce chapitre.

Au coeur du débat se trouvera ce qui fait la différence entre la mathématique et l'informatique : en informatique, il ne suffit pas d'affirmer l'existence théorique d'un objet, il ne suffit même pas d'en trouver une démonstration constructive c'est-à-dire un algorithme, encore faut-il tenir compte des contraintes d'espace et de temps imposées par le monde dans lequel nous vivons. Il faut que la solution soit calculable avec une place mémoire et un temps raisonnables pour l'homme ou la machine.

Comme une méthode est habituellement conçue pour résoudre plusieurs énoncés d'un même type avec des données différentes, la mesure naturelle de la qualité de cette méthode doit reposer sur la résolution du **plus mauvais** cas parmi tous les cas susceptibles d'être soumis.

D'un point de vue très concret, c'est le nombre total d'opérations élémentaires qui sera finalement déterminant.

Mais, afin de comparer ce qui est comparable, il faut ramener ce nombre d'opérations à l'ordre de grandeur du problème traité c'est-à-dire à la taille des données.

La complexité d'une procédure est ainsi la **borne supérieure du nombre d'opérations qu'elle requiert exprimée en fonction de la taille de l'énoncé d'entrée.**

La complexité d'un problème est la complexité du meilleur algorithme connu pour le résoudre.

Cette complexité dépend donc de l'état de l'art !

Deux questions fondamentales se trouvent du même coup posées :

- 1) Jusqu'où peut-on améliorer un algorithme donné ? Connait-on des méthodes de complexité optimale dans certains cas et existe-t-il de nombreux problèmes qu'on sait bien résoudre, c'est-à-dire dont la complexité est faible (par exemple n ou n^2 , si n est la taille à l'entrée) ?
- 2) Les complexités induisent-elles un regroupement des problèmes en classes et donnent-elles des méthodes pour passer d'une classe à une autre, d'un problème « difficile » à un problème plus « facile » ?

Les réponses à ces deux questions, ou plutôt groupes de questions, vont être données et expliquées.

Ces réponses laissent peu d'espoir à l'algorithmique « classique » de résoudre beaucoup de choses.

Elles justifient du même coup toutes les recherches en intelligence artificielle.

4.3.1 Trois classes de problèmes :

Les meilleurs algorithmes sont les algorithmes linéaires, c'est-à-dire de complexité en $(a \times n = b)$, dits encore en $O(n)$, si n est la taille de l'entrée.

De tels algorithmes linéaires existent effectivement. Par exemple, le calcul de la somme de 2 nombres de n_1 et n_2 chiffres requiert, par la méthode que chacun apprend à l'école primaire, le calcul d'au plus $n_1 + n_2$ nombres de un chiffre et de $\max(n_1, n_2)$ retenues. La multiplication de deux nombres de un chiffre est une opération élémentaire que toute machine sait exécuter, l'addition d'un nombre et d'une retenue également.

Ainsi, l'algorithme d'addition possède au total une complexité en $O(n_1 + n_2)$, étant entendu que cette expression n'est qu'un ordre de grandeur où l'on ne tient pas compte des facteurs constants (d'où ici la disparition des calculs liés aux retenues).

D'autres algorithmes bien connus - division, extraction de racines carrées, solutions d'une équation du second degré - viennent tous tomber, par généralisation de cette linéarité, dans la première classe de problèmes : les algorithmes polynomiaux ; algorithmes qui ont été décrits dans ce chapitre jusqu'au paragraphe 4.2.

1. Les algorithmes « polynomiaux » : la classe P

On appelle « *bon* » problème - ou *problème appartenant à P* - tout problème pour lequel on connaît un algorithme de complexité égale à un **polynôme de degré constant**, connu, indépendant de la taille d'entrée n .

Mais d'autres problèmes sont plus difficiles.

2. Les problèmes « exponentiels par nature » : la classe E

Tout problème dont la complexité est a priori au moins en f^n (f étant une constante ou encore un polynôme en n) est réputé « exponentiel » par nature. Il en est ainsi à chaque fois que le nombre de sorties demandées est lui-même, déjà, exponentiel.

Ceci arrive, a priori, pour un problème où l'on demande de construire tous les sous-ensembles d'un ensemble, toutes les cliques (sous-graphes complets) d'un graphe,

ou encore tous les sous-arbres d'un graphe ; on montre en effet que le nombre de sorties est alors de la forme : (polynôme en n) à la puissance n . Quelques problèmes se comportent encore plus mal : leur complexité se révèle par exemple de la forme :

$$\begin{array}{c} 2^n \\ \vdots \\ 2 \\ 2 \end{array}$$

soit un empilement de m exponentielles de base 2, avec le dernier exposant lui-même égal à 2^n , où n mesure toujours la taille de l'entrée, id est des données.

Il en va ainsi de problèmes de reconnaissance de mots formés d'expressions régulières sur des langages correspondants à des alphabets et des règles de constructions relativement simples.

Notons que, pour de petites valeurs de n , un algorithme exponentiel peut être plus rapide qu'un algorithme polynômial.

Mais, c'est la différence profonde entre les problèmes, mise concrètement en évidence, de toute manière, pour des valeurs grandes de n , qui est le sujet de ce chapitre.

Nous parvenons ici au noeud de la question.

Il existe, en effet, des problèmes que l'on ne peut classer a priori ni dans l'une ni dans l'autre des classes précédentes : P et E.

Ces problèmes sont innombrables. Ils n'impliquent jamais dans leurs énoncés une énumération par essence exponentielle. Nous ne connaissons pourtant pas actuellement (1986) et ce, pour beaucoup d'entre eux, un algorithme efficace (polynômial).

3. Les problèmes ni polynomiaux, ni exponentiels par nature : la classe III

Dans cette classe, on trouve :

- la résolution des systèmes d'équations en variables entières [Diophante (410)].
- la détermination d'un circuit passant par tous les sommets d'un graphe (hamiltonien) [Hamilton (1870)].
- l'existence d'une famille choisie parmi des sous-ensembles donnés qui recouvre un ensemble donné.
- la construction d'emplois du temps (respectivement de colorations) respectant des contraintes (respectivement des relations binaires).
- l'existence d'un ensemble de valeurs logiques qui permettent de rendre VRAIE la valeur d'une expression logique quelconque donnée [Cook (1971)].
- l'optimisation du trajet d'un voyageur de commerce à travers un réseau de villes.
- la sélection des fichiers à interroger dans une banque de données informatisée pour obtenir une information au moindre coût.
- la localisation de centres de services (téléphone, télévision, urgences) pour un maximum de clients et un minimum de centres.
- le chargement optimal d'un équipement (sac à dos, train, bateau, avion) au moindre coût.
- la découpe optimale (papier, carton, barres d'acier, pièces de fonderie, trajets dans l'espace aérien, choix d'investissements, machines-outils, systèmes d'exploitation informatiques).
- le diagnostic (pannes, maladies, circuits imprimés).

La liste est longue et pourrait continuer encore.

Dès lors, la problématique est la suivante : peut-on espérer classer certains de ces problèmes dans la famille P ou dans la famille E ?

La réponse actuelle, après des dizaines ou quelquefois des centaines d'années d'efforts, est qu'aucun algorithme polynômial n'a été trouvé pour aucun de ces problèmes :

CETTE DERNIERE CLASSE EST UNE ENIGME.

Bien plus, tous les problèmes précédents ont été prouvés équivalents dans le sens suivant.

Si quelqu'un est à même d'exhiber, aujourd'hui ou plus tard, un bon algorithme (polynômial) pour l'un quelconque de ces problèmes, alors tous seront automatiquement résolus du même coup de bonne manière, c'est-à-dire par un algorithme également polynômial.

Dans la suite de ce chapitre, nous allons donc réfléchir sur cette étrange classification des problèmes et l'étayer par nombre de considérations qui concernent à la fois la classe P - polynômiale - et la classe III - énigmatique.

La classe P la plus sympathique a déjà été traitée au paragraphe IV.2 : c'est l'ensemble des problèmes polynômiaux.

La seule question est, comme on l'a vu alors, qu'ils sont très peu nombreux : les problèmes formels que l'homme sait aujourd'hui bien résoudre (id est de façon algorithmique et polynômiale) sont une toute petite famille. Tous les autres sont difficiles et **ils relèvent tous, par définition, de l'intelligence artificielle !...**

Remarquons enfin que même lorsque, pour un énoncé en langage courant, un algorithme polynômial existe, le problème de reconnaître cela, de formuler, de modéliser l'énoncé pour le rendre justiciable du bon algorithme est encore lui-même, fondamentalement, un problème non polynômial !

4.4 La classe des problèmes NP : non déterministes polynômiaux

4.4.1 Deux familles de machines

Quand, pour résoudre un problème, on ne dispose ni d'une formule explicite, ni d'une expression récursive de complexité convenable, il reste les deux dernières méthodes : construction d'un algorithme effectif de calcul ou énumération.

Dire que la complexité d'un problème est polynômiale c'est dire que l'on est parvenu en un nombre polynômial d'étapes, et, en tous cas, sans énumération à résoudre le problème.

Le modèle abstrait pour un tel algorithme polynômial est une boîte noire qui ne connaît qu'un ensemble donné d'opérations élémentaires : $+$, $-$, \times , $/$, OU, ET, LIRE, ECRIRE, SI... ALORS..., REPETER. Cet *automate* se trouve, à un moment donné dans un état bien déterminé, il fait une seule chose à la fois qui dépend de cet état. Il passe alors à un autre état et tout recommence. On parle encore ici de *machine de Turing déterministe (MTD)*.

L'introduction - dont nous avons fondamentalement besoin dans certains problèmes - d'un schéma énumératif, c'est-à-dire de tâtonnements, d'essais provisoires, se laisse

commodément modéliser par un autre modèle de machine de Turing (tout aussi abstraite) dite *non déterministe* (*MTND*).

Dans une telle machine, il existe en plus du jeu d'instructions habituel, une instruction spéciale « CHOIX [E] », qui a pour effet de créer autant de copies de l'état courant de la machine qu'il y a d'éléments dans l'ensemble E . Par convention, une telle machine s'arrête dès que l'une des copies parvient à l'instruction FIN.

Exprimons, dans ce nouveau langage, deux algorithmes non déterministes (AND) pour deux problèmes simples.

Problème 1 : Faisabilité d'une expression logique

Une expression logique E est donnée. Elle comporte les symboles \vee (OU), \wedge (ET), \neg (NON), et des variables logiques, q_1, q_2, \dots, q_n . On cherche à trouver un ensemble de valeurs pour les q_i tel que, pour cet ensemble, la valeur de l'expression $E(q_1, q_2, \dots, q_n)$ soit VRAI.

Par exemple, si $E = (q_1 \vee \neg q_2 \vee \neg q_3) \wedge (q_1 \vee q_2 \vee q_4) \wedge (q_3 \vee \neg q_4) \wedge (\neg q_1)$, l'ensemble (FAUX, VRAI, FAUX, FAUX) convient.

L'algorithme, pour une MTND, est très simple à écrire :

```

REPETER POUR  $i \in [1 : n]$ 
|    $q_i \leftarrow \text{CHOIX } [\text{VRAI}, \text{FAUX}]$ 
FR i
    Si  $E(q_1, q_2, \dots, q_n)$  ALORS succès ; FIN.
    SINON échec.

```

Cet algorithme crée donc $2^n = 16$ copies de lui-même et la douzième copie, en particulier, conduira à un succès. L'expression E est alors dite « *faisable* ». **Problème 2 :** Coloriage d'une carte de géographie en 3 couleurs.

Il s'agit de colorier les pays P_1, P_2, \dots, P_n d'une carte plane avec seulement 3 crayons de couleurs différentes, en faisant en sorte que deux régions limitrophes soient coloriées différemment.

L'algorithme non déterministe est :

```

couleur ( $P_1$ )  $\leftarrow c_1$  {colorier un pays ainsi que les pays adjacents}
REPETER POUR  $i \in [1 : n]$ 
|   Si  $P_1$  adjacent à  $P_i$  ALORS couleur ( $P_i$ )  $\leftarrow c_2$ 
FR i
    REPETER POUR  $k \in [1 : n]$  TANT QU'il existe un pays non colorié
        |   Si le pays  $P_k$  n'est coloriable qu'avec la couleur  $j$ 
        |       ALORS couleur ( $P_k$ )  $\leftarrow j$ 
    FR k
    Si tous les pays sont colorés ALORS FIN.
    SINON soit  $P_{MIN}$  le pays de plus petit indice disposant de deux
        couleurs  $j_1$  et  $j_2$  :
        couleur ( $P_{MIN}$ )  $\leftarrow \text{CHOIX } [j_1, j_2]$ 

```

On voit ici avec quelle simplicité une MTND permet d'écrire les algorithmes d'énumération. Elle libère le programmeur de toute la gestion des retours en arrière (du « backtrack »). Tout se passe comme si les copies, qui sont finalement chacune des machines déterministes, travaillaient dans un parallélisme total sans problème de faisabilité technique. Le nombre de ces machines déterministes est, en effet, tout à fait

quelconque. Il « explose » très vite, puisqu'il est par nature exponentiel (voir les deux cas précédents).

Mais, peu importe, nous touchons le point important et deux remarques viennent tout de suite à l'esprit.

Remarque 1 : Dans chacun des cas précédents, non seulement chaque copie, prise isolément, est déterministe, mais, en outre, le nombre d'étapes, autrement dit la complexité de chaque copie, est polynômial (en fait, linéaire dans les deux cas précédents). Par généralisation, nous appelons *algorithme non déterministe polynômial*, *NP* en abrégé, tout algorithme qui peut être exécuté en un temps polynômial sur une machine de Turing non déterministe. Notons bien que NP n'est pas l'abréviation de non-polynômial, mais, au contraire, de non déterministe et polynômial.

Corollaire immédiat : $P \subset NP$

L'ensemble des problèmes polynômiaux sur une MTD appartient à NP, puisqu'une MTD est un cas particulier de MTND.

Remarque 2 : La deuxième remarque tient au fait, qu'a priori, il s'en faut d'un cheveu pour passer de *NP* à *P*. Ainsi :

- dans le second problème, si l'on s'autorise seulement 2 couleurs, l'algorithme précédent donne la réponse (possible, impossible) sans jamais utiliser la boucle REPETER, où figure l'instruction cruciale de CHOIX. La conclusion est obtenue en $O(n)$ étapes, sans aucun retour en arrière : la MTND est réduite strictement à une MTD.
- dans le premier problème l'ensemble des valeurs possibles est déjà restreint à deux éléments. Mais, ce qui joue, dans ce cas, pour bien maîtriser l'énumération, est le nombre de variables logiques à l'intérieur d'une même parenthèse. Supposons, en effet, que ce nombre soit au plus égal à deux et que l'expression à évaluer se présente sous la forme normalisée :

$$\bigwedge_{i=1}^m (r_i \vee s_i)$$

dans laquelle r_i et s_i désignent soit un q_j , soit un $\neg q_j$. Dès lors, quand je fixe la valeur du premier terme, soit r_i de la i^{ime} parenthèse, alors, ou bien celle-ci prend du même coup la valeur VRAI et je passe à la $(i+1)^{ime}$, ou bien ce n'est pas le cas ; mais alors, la forme du second terme, soit s_j , me donne directement - c'est-à-dire sans CHOIX -, par propagation directe de contrainte, la valeur que doit prendre la variable associée. Si ce processus ne rencontre aucune contradiction jusqu'à $i = n$, il fournit simultanément un vecteur (q_1, q_2, \dots, q_n) solution, sinon l'expression donnée n'était pas faisable.

Ainsi donc, la séparation entre *P* et *NP* se produit, brusquement, pour des valeurs très proches : $n = 2$ et $n = 3$.

La question est donc maintenant de savoir si cette situation est effectivement irrémédiable.

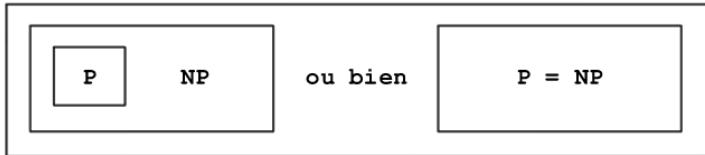
Enoncé d'une autre manière, ceci devient encore :

Les machines non-déterministes sont-elles définitivement plus puissantes (c'est-à-dire peuvent-elles résoudre plus de problèmes que les machines déterministes ?)

Ou encore, plus succinctement : *P* est-il égal à *NP* ?

A-t-on :

La réponse est : **actuellement nous ne savons pas.**



4.5 Liste de problèmes dans NP

Une liste de quelques problèmes de NP est très facile à établir. On y trouve les problèmes suivants :

1. Faisabilité d'une expression logique.
2. 3-coloration dans un graphe.
3. Construction d'une clique de k sommets dans un graphe non orienté.
4. Recouvrement d'ensemble : étant donnée une famille F de parties E_i d'un ensemble E trouver une sous famille G de F , telle que :

$$\bigcup_F E_i = \bigcup_G E_i$$

5. Partition d'ensemble ; même énoncé que le précédent mais on impose en outre à l'arrivée :

$$E_i \cap E_j = \emptyset \text{ pour tout } E_i \text{ et } E_j \text{ dans } G$$

6. Existence d'un circuit hamiltonien dans un graphe non orienté
7. Solution du sac à dos ; déterminer une solution en variables x_j binaires à l'équation :

$$\sum a_j x_j = b, \text{ } a_j \text{ et } b \text{ entiers donnés}$$

Plus généralement solution d'une équation diophantienne (id est en nombres entiers) quelconque.

8. Partage en deux ; n nombres y_j de \mathbb{N} étant donnés, les partager exactement en deux sous-ensembles I_1 et I_2 tels que :

$$\sum_{j \in I_1} y_j = \sum_{j \in I_2} y_j$$

9. Existence d'un circuit d'un voyageur de commerce dans un graphe orienté, de coût total inférieur à b donné.

Cette liste pourrait être allongée démesurément : **la plupart des problèmes courants appartiennent en effet à NP .**

Il faut d'ailleurs ne pas oublier que les problèmes cités sont des problèmes modèles. Il correspond à chacun d'eux plusieurs énoncés réels comme : ordonnancement de tâches, affectation de personnels, optimisation de transports, de raffinage, de localisation d'usines, gestion de production, conception en électronique ou en architecture.

Pour montrer qu'un problème de cette liste est dans NP , il suffit d'écrire un algorithme non déterministe qui le résout avec une complexité polynomiale.

Par exemple, encore pour le problème 9 :

[Existence d'un circuit hamiltonien de coût $\leq b$; dans un graphe G orienté]

```

 $V_1 \leftarrow 1$  {ville de départ}
 $S \leftarrow \{2, \dots, n\}$  {ensemble des sommets à visiter}
 $coût \leftarrow 0$  {coût total}
 $nv \leftarrow 1$  {nombre total de villes déjà visitées}

REPETER TANT QUE successeur  $(v_{nv}) \neq$  vide
| {successeur  $(v_{nv})$  désigne l'ensemble autorisé (pas de circuit parasite)
| des villes qu'il est possible d'atteindre à partir de  $(v_{nv})$ }.

|  $(v_{nv-1}) \leftarrow$  CHOIX (successeur  $(v_{nv})$ )
|  $nv \leftarrow nv + 1$ 
|  $coût \leftarrow coût + coût(\text{arc } (v_{nv-1}, v_{nv}))$ 

FR
Si  $nv = n$  ET  $coût \leq b$  ALORS succès; FIN
SINON échec.

```

Chaque copie de cet algorithme comportera bien un nombre polynômial d'étapes, à savoir exactement n , et ceci indépendamment de la valeur de b ; donc ce problème de l'existence d'un circuit de coût inférieur à b est dans la classe NP .

Note : Par contre le problème de circuit hamiltonien **optimal** n'est pas dans NP ; en effet il comporte comme sous-problème la preuve de l'inexistence de circuits de coûts supérieurs à b . Or, ce problème n'est pas dans NP .

Ce problème est le complémentaire du problème 9, c'est-à-dire que l'on recherche maintenant les circuits qui ne sont pas des solutions du problème 9.

Mais, il est faux de dire que si un problème Q est dans NP alors le problème complémentaire \bar{Q} est dans NP .

Ce résultat serait conforme à « l'intuition » c'est-à-dire à ce qui se passe pour les problèmes de la classe P . Pour ceux de NP au contraire, passer au complémentaire revient à échanger dans les AND toutes les occurrences de succès et d'échecs. Mais un succès terminal pour Q signifie les échecs de toutes les branches de l'AND de départ. Il faut donc attendre, pour obtenir la réponse, que toutes les MTD aient fini leur travail : les complexités s'additionnent et comme le nombre des MTD n'est pas nécessairement polynômial rien n'assure que $\bar{Q} \in NP$ lorsque $Q \in NP$.

En particulier, dans le cas du problème : « Existe-t-il un circuit hamiltonien dans G de coût $\geq b$ », on ne sait pas écrire un AND pour ce problème, qui n'est donc pas classé dans NP , et ceci est essentiellement dû à ce que l'écart entre b et le coût réel d'un circuit hamiltonien peut, a priori, être aussi grand que l'on veut (non borné par un polynôme fonction du nombre de villes à visiter).

Le problème du circuit optimal pour un voyageur de commerce n'est donc pas classé dans NP .

La plupart des problèmes d'optimisation sont dans le même cas : seule la recherche de l'existence d'une solution de coût donné est dans NP , l'optimisation elle-même n'est pas classée aujourd'hui.

4.6 Etude des problèmes de NP par classes d'équivalence

Dans un premier temps, nous allons montrer que nombre de problèmes d'énoncés très différents, mais dans NP , sont en fait équivalents. Dans un second, nous montrerons que, vus la taille de la classe ainsi définie et les efforts actuels des informaticiens et mathématiciens, **il reste peu de chances pour que $P = NP$.**

Les définitions qui suivent vont d'abord permettre de préciser ce qu'il convient d'entendre par problèmes équivalents.

– **Définition 1 :**

Un problème Q est « *réductible* » à un problème R si et seulement si, pour toute solution s de R , il existe une fonction $g(s)$, calculable de façon polynômiale, telle que $g(s)$ soit solution du problème Q .

Q est alors par définition un « cas particulier » de R et l'on écrit :

$$Q \Rightarrow R$$

Ceci signifie encore : si je sais résoudre R , alors je sais résoudre Q .

– **Définition 2 :**

Lorsqu'on a, à la fois, $Q \Rightarrow R$ et $R \Rightarrow Q$, alors, on dit que Q et R sont équivalents.

– **Définition 3 :**

Un problème est réputé NP -difficile si et seulement si tout problème de NP lui est réductible.

Note : Attention, *NP-difficile* n'impose donc pas l'appartenance à NP .

Cette définition surprenante est justifiée par le théorème suivant qui sert de point de départ pour établir de nombreuses équivalences entre problèmes.

4.7 Le théorème fondamental (Cook (1971))

Le problème de la faisabilité d'une expression logique est NP -difficile.

Ce théorème affirme donc que la solution de tout problème de NP - c'est-à-dire de tout problème traitable par une machine de Turing non déterministe en temps polynômial - peut se déduire à l'aide d'une transformation elle-même polynômiale, de la solution du problème de la faisabilité.

On peut dire encore : pour résoudre un problème quelconque de NP , il suffit de savoir résoudre le problème de la faisabilité.

Tout $NP \Rightarrow$ faisabilité.

Le résultat clé ci-dessus a d'abord été obtenu par Cook à propos de démonstration automatique de théorèmes.

Il donne un résultat plus fin que les théorèmes désormais classiques d'indécidabilité dans la plupart des systèmes formels (Church (1932), Gödel (1931), cf. chapitre 3).

La preuve de ce théorème de Cook, que nous donnons maintenant, est un peu longue, car elle nous oblige à préciser le modèle formel d'une MTND.

Une machine de Turing non déterministe est caractérisée par une tête de lecture écriture, une bande d'enregistrement divisée en cases où sont inscrits des symboles : $\sigma_1, \sigma_2, \dots$ ou σ_r , un ensemble fini F d'états internes et par sept propriétés de a) à g).

- a) la tête a accès à une case de la bande exactement à chaque instant ;
- b) à tout moment, chaque case contient un symbole unique ;
- c) un seul état est autorisé à la fois ;
- d) seule la case sous la tête peut être modifiée ;
- e) les autres actions : déplacement de la tête, changement d'état interne, instruction CHOIX ne sont possibles que par la fonction Def_M de définition de la machine M ;
- f) l'état initial est un état possible ;
- g) l'état associé au premier succès (respectivement au dernier échec) est appelé état final.

Etant donné une MTND décrite comme ci-dessus, nommée M , et par ailleurs un problème de taille n quelconque, nommé Q , que cette machine peut résoudre, nous allons construire explicitement, une expression logique $E_{M,Q}$ avec :

$E_{M,Q}$ est faisable $\iff Q$, est soluble sur M en un temps polynômial

Le point important dans cette preuve est qu'un algorithme polynômial permet toujours la démonstration de la faisabilité de $E_{M,Q}$.

- Q soluble sur M en un temps polynômial signifie qu'il existe au moins une séquence d'états : $(e_0, e_1, e_2, \dots, e_m)$

- avec - $e_i \in Def_M(e_i - 1)$ { e_i est un « coup légal »} ;
- e_0 est l'état initial ;
- e_m est l'état final tel que : $m \leq p(n)$ où p est un certain polynôme ;
- aucune autre chaîne légale d'états e_0, e_1, \dots, e_k n'a plus que $p(n)$ cases, c'est-à-dire n'a pas plus de $p(n)$ étapes.

L'expression $E_{M,Q}$ que nous allons maintenant effectivement construire va simuler l'enchaînement e_0, e_1, \dots, e_m .

Dans $E_{M,Q}$ vont intervenir des variables. Chaque affectation de valeurs à cet ensemble de variables va correspondre à au plus une séquence légale d'états.

Finalement, $E_{M,Q}$ prendra la valeur VRAI si et seulement si une affectation existe qui correspond à une séquence conduisant à un « succès ».

Les variables utilisées sont celles qui permettent l'exacte description de la MTND considérée ; elles sont de trois sortes :

- $t(i, \theta) = 1$ si et seulement si la tête de lecture écriture pointe sur la case i au temps θ , ($1 \leq i \leq m, 1 \leq \theta \leq n$).
- $u(j, \theta) = 1$ si et seulement si M est au temps θ dans l'état (unique) e_j .
($1 \leq j \leq f$ où f est le nombre d'états possibles dans F et $1 \leq \theta \leq m$).
- $v(i, k, \theta) = 1$ si et seulement si, au temps θ la i -ème case sur la bande lecture de M contient le k -ième symbole ; on a : ($1 \leq i \leq m, 1 \leq k \leq r$ et enfin $1 \leq \theta \leq m$).

Le nombre de symboles r est fixe, indépendant de n , si bien que le nombre de variables logiques introduites est de l'ordre de m^2 , c'est-à-dire un polynôme en fonction du problème à l'entrée de taille n .

Nous allons maintenant faire correspondre aux sept propositions caractéristiques de M : a), b), ..., g), sept expressions logiques : A), B),..., G) :

A. « **La tête examine une cellule à la fois** ».

$$\forall \theta, 1 \leq \theta \leq m$$

$$\sum_{i=1}^m t(i, \theta) = 1 \text{ donc au total } \prod_{\theta=1}^m \sum_{i=1}^m t(i, \theta) = 1$$

Cette expression contient exactement m termes ; la condition a) elle-même s'écrit donc au moyen de m^2 termes.

B. « **A tout moment il y a un symbole unique par case** ».

$$\forall \theta, 1 \leq \theta \leq m$$

$\forall i, 1 \leq i \leq m$, i désigne la case numéro i , il vient :

$$\sum_{k=1}^r v(i, k, \theta) = 1$$

Comme r est indépendant de n , la condition b), s'écrit à l'aide de m^2 termes.

C. « **M ne prend qu'un état à la fois** ».

$$\forall \theta, 1 \leq \theta \leq m$$

$$\sum_{j=1}^f u(j, \theta) = 1$$

La condition C s'écrit en m termes, puisque le nombre d'états, f , ne dépend que de la machine M .

D. « **Sur la bande, seule la case sous la tête peut être modifiée** ».

$$\forall i, 1 \leq i \leq m$$

$$\forall k, 1 \leq k \leq r$$

$$\forall \theta, 1 \leq \theta \leq m$$

$(v(i, k, \theta) = 1) \Leftrightarrow ((t(i, \theta) = 1) \vee (v(i, k, \theta + 1)) = 1)$ c'est-à-dire si la i -ème case est sous la tête de lecture de M , avec le symbole k , c'est que :

- soit la tête lit la i -ième case au temps θ ,
- soit la i -ième case contient encore le k -ième symbole à la date $\theta + 1$.

Ceci est bien cohérent avec les conditions déjà traduites en particulier A et B. La condition D s'écrit au total avec $O(m^2)$ symboles puisque r est constant.

E. **Chaque nouvel état ne peut être obtenu que par la fonction Def_M de définition de M , c'est-à-dire au temps θ :**

Si la i -ème case contient le symbole j ,

Si la tête lit la case i , et M est dans l'état k

ALORS l'état suivant est l'un des états $Def_M(i, j, k)$

Nous écrirons cette proposition, du type $x \Rightarrow y$, sous la forme $\neg x \vee y$ soit, ici, après traduction des trois prémisses :

$$(v(i, k, \theta) = 0) \vee (u(j, \theta) = 0) \vee (t(i, \theta) = 0) \vee [v(i, k_l, \theta + 1) = 1 \wedge u(j_l, \theta + 1) = 1 \wedge t(i_l, \theta + 1) = 1].$$

où l désigne l'un quelconque des états de $Def_M(i, j, k)$.

Comme M est non déterministe, il existe pour un triplet (i, j, k) plus d'un triplet $Def_M(i, j, k)$, mais il en existe toujours un nombre fini.

Ainsi, pour i, j et k fixés, la proposition ci-dessus est de longueur bornée indépendamment de n . Comme le nombre f d'états est lui-même constant, la condition E , s'écrit au total au moyen de $O(m^2)$ termes.

F. « **L'état initial est légal** », c'est-à-dire, en repérant par l'indice 1 l'état à l'instant 1, la case la plus à gauche étant elle-même numérotée 1 :

$$(u(1,1) = 1) \wedge (t(1,1) = 1)$$

En outre, les cases de 2 à $n + 1$ contiennent l'énoncé de départ, Q , et les autres sont vides ; soit, si le symbole blanc est codé 1 :

$$\left(\bigwedge_{2 \leq i \leq n+1} (v(i, j_i, 1) = 1) \right) \left(\bigwedge_{n+2 \leq i \leq m} (v(i, 1, 1) = 1) \right)$$

La condition F est donc de longueur $O(m)$.

$$F \equiv ((u, 1, 1) = 1) \wedge (t, 1, 1) = 1) \left(\bigwedge_{2 \leq i \leq n+1} v(i, j_i, 1) = 1 \right) \left(\bigwedge_{n+2 \leq i \leq m} v(i, 1, 1) = 1 \right)$$

G. g) affirme qu'il existe finalement un état, soit Λ qui est l'état cherché, égal au succès.

En admettant, qu'à partir de ce moment, M reste dans cet état (en particulier M sera dans l'état Λ au temps $m = p(n)$), il vient pour g) : $u(\Lambda, m)$.

La condition complète, pour la résolution par M du problème Q , s'écrit donc au bout du compte :

$$H = A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G$$

Cette condition s'exprime, par construction, à l'aide d'au plus $O(m^3)$ termes et :

La condition H est faisable si et seulement si M résout le problème Q .

En effet, si une séquence d'état e_1, e_2, \dots, e_m conduit à e_Λ , alors, par là-même, les affectations des variables t, u, r sont définies, qui donnent à H la valeur VRAI.

Mais, inversement, si ces affectations sont connues, elles correspondent via la fonction Def_M à une séquence d'états légaux.

Puisque nous n'avons imposé aucune restriction à notre machine non déterministe M , nous avons ainsi montré que **tout problème soluble sur M (tout langage reconnaissable par M) est réductible (de façon polynômiale) au problème de la faisabilité d'une expression logique**.

4.8 La classe des NP -complets

A partir de ce théorème de Cook, toutes les réductions dans le sens « $Q \Rightarrow$ faisabilité » sont donc du même coup établies pour tout $Q \in NP$.

La relation de réductibilité est par ailleurs transitive. Si l'on parvient maintenant à montrer qu'inversement la faisabilité est réductible à un problème Q_1 , lui-même réductible à un problème Q_2 , et ainsi de suite, nous aurons une classe de problèmes équivalents.

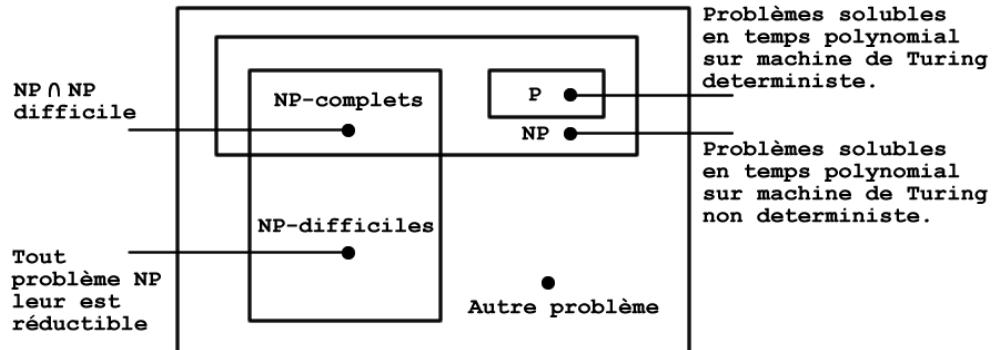
Ceci oblige en fait à préciser la classe III, énigmatique, en la coupant en deux parties, suivant qu'on a pu prouver ou non qu'un problème était ou non dans NP .

Définition.

Un problème est dit *NP-complet* si et seulement si :

- il est *NP-difficile* et
- il est dans *NP*.

Une figure semble indispensable pour bien voir où nous en sommes :



Le résultat $P = NP$ paraît aujourd’hui bien **improbable** : pour la plupart des problèmes classiques, pratiquement difficiles, l’appartenance à la classe *NP-difficile* a été prouvée.

Mais, ce qui est plus surprenant encore, c’est que, dans la presque totalité des cas, ces problèmes sont dans *NP*. Ils sont donc *NP-complets* et en ce sens tous équivalents :

Si l’on savait résoudre l’un d’entre eux en un temps polynômial, alors on saurait résoudre tous les problèmes de la classe dans un temps de même grandeur.

Rappelons, par ailleurs, les deux autres caractéristiques de ces problèmes *NP complets* :

- on ne sait pas si ces problèmes sont non polynomiaux ;
- on ne connaît pas d’algorithme (déterministe) borné polynomialement pour les résoudre.

Il reste à faire quelques démonstrations d’équivalence pour en montrer l’esprit général.

L’idée est très intéressante puisqu’elle consiste à ramener un problème de la classe à un autre. L’ennui est finalement qu’on ne sait pas bien résoudre le problème-clé : la faisabilité. Mais, les principes de modélisation et de transformation d’énoncés peuvent servir aussi bien dans les résolutions concrètes.

Il ne faut, en effet, jamais perdre de vue dans cette étude, que même si la preuve théorique de l’appartenance à la classe *NP-complet* d’un problème général Q est faite, tout énoncé particulier avec des données précisées peut malgré tout être résolu!...

La résolution en temps polynomiale de cet énoncé précis n’est même pas impossible.

La seule chose que dise la théorie, c’est qu’il n’existe pas **d’algorithme polynomiale général pour le cas général**.

4.9 Quelques démonstrations d'équivalences entre problèmes

L'équivalence entre une large famille de problèmes est actuellement acquise. La réductibilité au problème de Cook de ces problèmes appartenant au départ à NP , a été effectuée, soit directement, soit indirectement par transitivité, selon le schéma arborescent de la figure 4.9 :

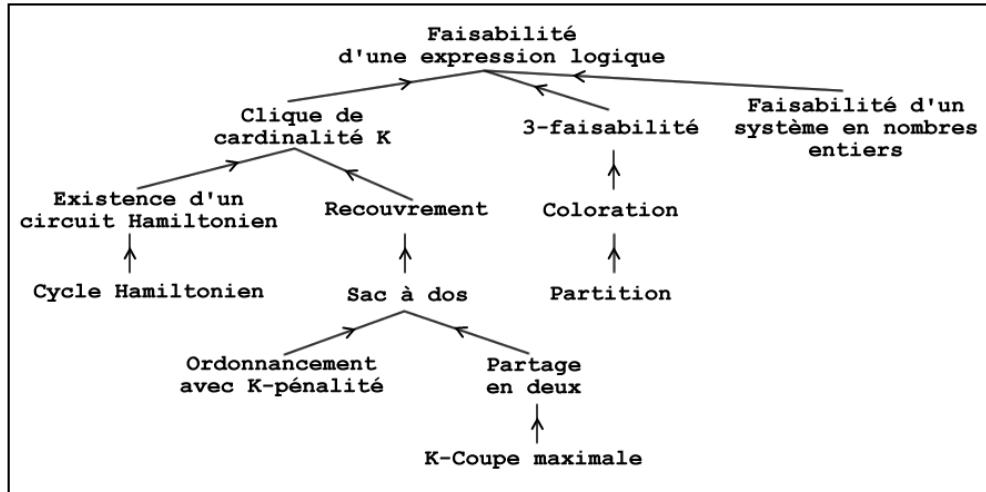


FIG. 4.9 – Quelques réductions établies entre NP-complets

Pour montrer qu'un problème Q est NP -complet, il nous reste à chaque fois à prouver que :

- 1) - Q est dans NP ;
- 2) - Q est NP -difficile, mais, comme Cook a prouvé que :

$Q \rightarrow$ Faisabilité pour tout Q de NP , il reste :
(la flèche signifiant « se réduit à »)
Faisabilité $\rightarrow Q$

Autrement dit, il faut et il suffit de construire une transformation polynômiale qui fait passer de Faisabilité à Q , telle qu'à toute solution de Q on puisse faire correspondre une solution de Faisabilité.

Montrons d'abord, en premier point, que la faisabilité d'expressions logiques, même formellement beaucoup plus simples que celles du cas général du théorème de Cook, est pourtant un problème bel et bien équivalent au cas général.

Exemple 1 :

- Le problème de la faisabilité d'une expression logique sous forme normale conjonctive (FNC) est NP -complet.
- Le problème de la faisabilité d'une expression logique sous FNC et avec au plus trois littéraux par clause est NP -complet.

Une expression est sous FNC, lorsqu'elle s'écrit comme le produit logique (opération \wedge) de sommes (opérations \vee) de quantités élémentaires appelées littéraux : chaque littéral ne peut être que : un q ou un $\neg q$, si q désigne une variable logique. L'expression

de H dans la démonstration du théorème de Cook peut être standardisée en FNC. En effet, les sommes :

$$\sum_{i \in I} t(i, \theta) = 1$$

peuvent aussi s'écrire, en passant des variables en 0 – 1 à des variables logiques (avec $1 \leftrightarrow$ VRAI et $0 \leftrightarrow$ FAUX) :

$$(\bigvee_{i \in I} t(i, \theta)) \bigwedge_{(i,j) \in I^* I, i \neq j} (\neg t(i, \theta) \vee \neg t(j, \theta))$$

La première parenthèse - ou clause - exprime qu'au moins une des variables est vraie ; la seconde clause affirme que pour chacun de $|I| \times \frac{(|I|-1)}{2}$ couples, il n'y a jamais 2 variables vraies simultanément.

Ainsi, les conditions A , B et C sont sous FNC.

La standardisation de D s'obtient en changeant, pour chaque occurrence du symbole \iff :

$$x \iff y \text{ en } (\neg x \vee y) \wedge (x \vee \neg y)$$

d'où une expression sous FNC, où la longueur totale a , au pire, été doublée.

Les expressions E , F et G deviennent sous FNC quand on change le type des variables de booléen (0 – 1) en logique (VRAI, FAUX).

Au total, la longueur de H a été multipliée par un facteur constant et le résultat de Cook reste valable pour les FNC.

Nous pourrons donc dans toute la suite nous restreindre à cette famille d'expressions FNC.

Parmi elles, les premières, les plus simples à être candidates pour le « clan » NP sont les 3-FNC, c'est-à-dire celles qui possèdent au plus 3 littéraux par clause.

1) - La faisabilité d'une 3-FNC est dans NP puisqu'il est évident d'établir un algorithme pour une MTND avec un nombre d'étapes constant et égal à 3 :

$\left[\begin{array}{l} \text{REPETER TANT QU'il existe une clause } c_i \text{ non satisfaitte} \\ \quad \{c_i \equiv f(x, y, z)\} \\ \quad x \leftarrow \text{CHOIX [VRAI, FAUX]} \\ \quad y \leftarrow \text{CHOIX [VRAI, FAUX]} \\ \quad z \leftarrow \text{CHOIX [VRAI, FAUX]} \\ \quad \text{Si } c_i = \text{FAUX ALORS échec} \\ \quad \text{FR} \\ \quad \text{Si toutes clauses satisfaites ALORS succès ; FIN} \end{array} \right]$

2) - Faisabilité \rightarrow (3 – Faisabilité)

La transformation suivante permet de passer de toute clause comportant i littéraux à une 3-clause :

$(x_1 \vee x_2 \vee \dots \vee x_i)$ devient, après introduction de $(i-3)$ nouvelles variables y :

$$(x_1 \vee x_2 \vee y_1) \wedge (x_3 \vee \neg y_1 \vee y_2) \wedge (x_4 \vee \neg y_2 \vee y_3) \wedge (x_5 \vee \neg y_3 \vee y_4) \dots \\ (x_{i-2} \vee \neg y_{i-4} \vee y_{i-3}) \wedge (x_{i-1} \vee x_i \vee \neg y_{i-3}).$$

Ces deux expressions sont formellement équivalentes comme on le montre aisément par induction sur i .

Donnons-nous une solution en x . Supposons $x_{i_0} = \text{VRAI}$.

Posons, alors, $y_j = \text{VRAI}$ pour $j \leq i_0 - 2$,
et $y_j = \text{FAUX}$ pour $j > i_0 - 2$.

L'expression en x et y a alors bien la valeur VRAI.

Inversement, si il existe un jeu de valeurs pour les y_j qui lui donne la valeur VRAI, alors, nécessairement, un x_i au moins doit être VRAI : comme la valeur de la première clause doit être VRAI, il vient :

$\left[\begin{array}{l} \text{Si } y_1 = \text{FAUX ALORS } x_1 \text{ ou } x_2 \text{ est VRAI et c'est fini;} \\ \\ \text{SINON } \{y_1 = \text{VRAI}\} \\ \\ \text{Si } y_{i-3} = \text{VRAI ALORS } x_i \text{ ou } x_{i-1} \text{ VRAI et fin;} \\ \\ \text{SINON } \{y_i = \text{VRAI}; y_{i-3} = \text{FAUX}\} \\ \\ \text{ALORS nécessairement il existe un } \\ \\ i_0, i \leq i_0 \leq i_4 \text{ tel que :} \\ \\ y_{i_0} = \text{VRAI et } y_{i_0+1} = \text{FAUX;} \\ \\ \text{donc : } x_{i_0+2} = \text{VRAI.} \\ \\ \text{FSI} \\ \\ \text{FSI} \end{array} \right]$

Ainsi, par exemple avec $i = 4$ et la transformation indiquée, $(x_1 \vee x_2 \vee x_3 \vee x_4)$ devient $(x_1 \vee x_2 \vee y_1) \wedge (x_3 \vee x_4 \vee \neg y_1)$ qui sont équivalentes.

Comme la nouvelle expression contient exactement $(i-1)$ clauses de 3 littéraux chacune pour chaque clause de littéraux au départ ($i \geq 4$), le nombre total de termes de la nouvelle expression est moins que 3 fois l'ancien.

Finalement, on sait donc réduire en temps polynômial tout problème de faisabilité à une 3 - faisabilité sous FNC.

Exemple 2 : « Le problème de l'existence dans un graphe d'une K-clique est NP-complet ».

{clique = sous-ensemble de sommets tous reliés deux à deux,
K-clique = clique de cardinalité K.}

- 1) - Ce problème est bien *NP*, car, quel que soit un graphe, il existe un nombre C^K , donc polynômial, de sous-ensembles de K sommets exactement. Pour chacun d'eux, il suffit de vérifier l'existence d'une arête pour tout couple de sommets, il y a donc $O(K^2)$ tests à effectuer.
- 2) - Faisabilité se réduit à K clique.

Soit une expression quelconque E sous FNC comportant K clauses :

$$C_1 \wedge C_2 \wedge \dots \wedge C_k \text{ avec } C_i = Vx_{ij} \text{ et } x_{ij} = \text{variable ou } \neg \text{variable.}$$

Définissons à partir d'elle le graphe :

- dont chaque sommet correspond à un doublet (x, i) , où x est un littéral de la i -ème clause C_i de E ;
- pour lequel un arc relie (x, i) à (y, j) si et seulement si :
 - $i \neq j$, les deux sommets correspondent à deux clauses différentes ;
 - $x \neq \neg y$, les deux littéraux ne sont pas la négation l'un de l'autre.

Avant d'aller plus avant dans la démonstration, visualisons un exemple de tel graphe.
Si E est l'expression :

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_3 \vee \neg x_4) \wedge (\neg x_1)$$

le graphe résultant possède neuf sommets, autant que de littéraux dans F . Une arête relie deux littéraux venant de clauses différentes et dont les valeurs peuvent être simultanément VRAI. (Figure 4.10).

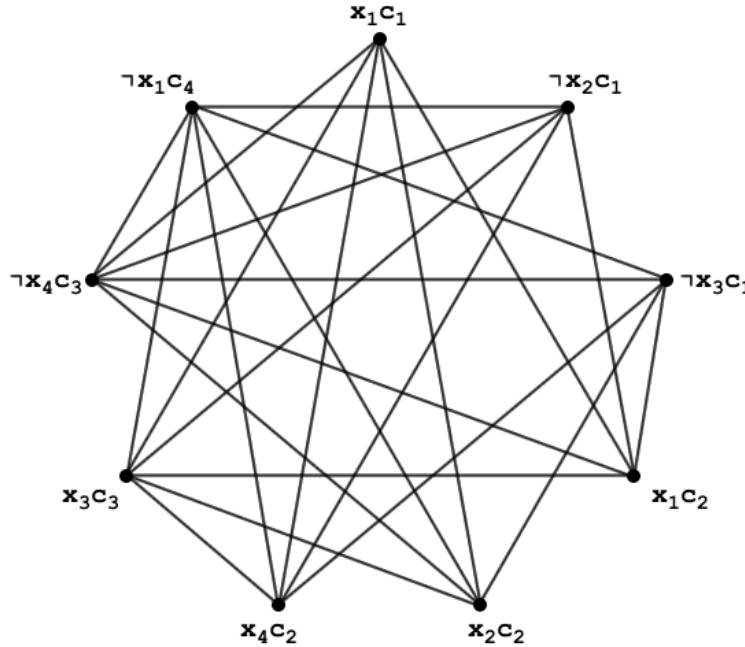


FIG. 4.10 – Graphe résultant pour le problème de la K-clique

L'existence d'une K clique dans ce graphe correspond donc à l'existence pour E d'un ensemble de K littéraux, appartenant à K clauses différentes, c'est-à-dire, ici, à toutes, et qui peuvent tous être VRAI en même temps.

Réciproquement, si l'expression est faisable, c'est-à-dire s'il existe un ensemble de valeurs de vérité pour les variables, tel que chaque clause soit VRAI, alors, il existe dans chacune des K clauses au moins un littéral, lui-même de valeur VRAI. A cet ensemble de littéraux correspond, par construction, dans le graphe, un ensemble de K sommets complètement reliés car associés à des clauses différentes et non deux à deux contradictoires, soit une K -clique.

En l'occurrence, avec l'expression donnée, $K = 4$ et on trouve par exemple la clique $(\neg x_3 C_1, x_2 C_2, \neg x_4 C_3, \neg x_1 C_4)$ ou bien encore la clique $(\neg x_2 C_1, x_4 C_2, x_3 C_3, \neg x_1 C_4)$, puisque de toute manière $(\neg x_1 C_4)$ est de la fête.

Exemple 3 : Le problème de l'existence d'une solution à un système d'équations en nombres entiers (S.N.E.) est NP-complet.

- 1) - Ce problème est NP , car il ne comporte qu'un nombre fini de choix différents (les valeurs des variables).
- 2) - Faisabilité → S.N.E. par transformation polynomiale :

Construisons, en effet, un S.N.E. de la forme : $Ax \geq b$ avec n inconnues booléennes x_j en correspondance biunivoque avec celles de l'expression E de départ ($x_j = 1$ dans S.N.E. $\leftrightarrow x_i = \text{VRAI}$ dans E) et avec K contraintes si E comporte K clauses ; posons pour les éléments a_{ij} de A ($1 \leq i \leq K$, $1 \leq j \leq n$) :

$$a_{ij} = \begin{cases} 1 & \text{ssi } x_j \text{ est un littéral de } C_i; \\ -1 & \text{ssi } \neg x_j \text{ est un littéral de } C_i; \\ 0 & \text{sinon.} \end{cases}$$

Dès lors, à la i -ème contrainte du S.N.E. correspond la i -ème clause, sous réserve de prendre pour second membre b_i de cette contrainte :

$$b_i = 1 - |\{j / \neg x_j \text{ est un littéral de } C_i\}|$$

Il est sûr que C_i sera satisfaite si et seulement si un littéral « positif » (correspondant à $a_{ij} = 1$) au moins est VRAI ou bien si un littéral « négatif » (soit $a_{ij} = -1$) au moins est FAUX.

Cette double condition s'écrit aussi dans le S.N.E. :

$$\sum_{j \in J_i^+} x_j - \sum_{j \in J_i^-} x_j \geq b_i$$

$$\begin{aligned} \text{avec } J_i^+ &= \{j / x_j \text{ est un littéral de } C_i\} \\ J_i^- &= \{j / \neg x_j \text{ est un littéral de } C_i\} \end{aligned}$$

Cette dernière inéquation s'écrit également, par définition des b_j :

$$\sum_{j \in J_i^+} x_j + |J_i^-| \geq 1 + \sum_{j \in J_i^-} x_j \quad (1)$$

Or, E satisfaite \Rightarrow un x_j au moins de $J_i^+ = 1$ ou bien
un x_j au moins de $J_i^- = 0$ pour tout $i, i \leq i \leq k$

En effet, si $x_{j_0} = 1$, avec j_0 dans J_i , entraîne bien (1) puisqu'en tous cas :

$$\sum_{j \in J_i^-} x_j \text{ est inférieur à } |J_i^-|$$

Dans l'autre cas, $x_{j_0} = 0$, avec j_0 dans J_i , suppose en premier lieu $|J_i^-| \geq 1$; comme la somme des x_j pour $j \in J_i^+$ est positive l'inégalité (1) est bien vérifiée puisqu'alors :

$$|J_i^-| - 1 \geq \sum_{j \in J_i^-} x_j$$

Ainsi, à toute solution du S.N.E., on peut faire correspondre, par la transformation polynomiale indiquée, une solution au problème de la faisabilité.

En particulier, pour la même expression E que dans l'équivalence précédente :

$$E = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_3 \vee \neg x_4) \wedge (\neg x_1)$$

le système en nombres entiers, ci-dessus défini, est :

$$\left\{ \begin{array}{lcl} x_1 - x_2 - x_3 & \geq -1 & \{\text{clause } C_1\} \\ x_1 + x_2 + x_4 & \geq 1 & \{\text{clause } C_2\} \\ x_3 - x_4 & \geq 0 & \{\text{clause } C_3\} \\ -x_1 & \geq 0 & \{\text{clause } C_4\} \end{array} \right.$$

Une résolution « directe » de ce S.N.E. conduit, d'après $\{C_4\}$, à : $x_1 = 0$; $\{C_1\} + \{C_2\}$ imposent alors :

$$x_4 - x_3 \geq 0$$

contrainte qui, avec $\{C_3\}$, donne finalement :

$$\begin{array}{ll} x_4 - x_3 = 0 \\ \text{Il reste } & -x_2 - x_3 \geq -1 \quad \{C_1\} \quad \text{d'où exactement deux solutions suivant la valeur} \\ \text{et} & x_2 + x_4 \geq 1 \quad \{C_2\} \\ \text{de } x_2 : & (0, 1, 0, 0) \text{ et } (0, 0, 1, 1) \end{array}$$

Ces deux solutions peuvent évidemment, à leur tour, être mises en correspondance avec les deux 4-cliques du problème précédent.

Chapitre 5

Les méthodes par propagation et énumération

Les méthodes énumératives – le type D de l'introduction du chapitre 4 – se trouvent donc être, et de loin, celles auxquelles il convient d'avoir recours pour la plupart des problèmes. Nous leur consacrons d'abord un nouveau paragraphe, puis une partie importante de la fin de ce chapitre 5. le chapitre 6 et enfin le chapitre 7 dans son entier.

5.1 Les méthodes de résolution par énumération

5.1.1 La classe des problèmes combinatoires

Un problème combinatoire relève par définition de la formulation générale :

“Chercher l'élément x de l'ensemble X satisfaisant un ensemble $K(x)$ de contraintes, où l'espace de recherche X contient un nombre fini de points distincts.”

C'est là une situation que l'on rencontre très souvent dans la réalité, elle concerne tous les cas où l'on a affaire à des éléments discontinus par nature : casse-tête logique, matériel à commander pour renouveler un stock, choix d'une décision parmi un ensemble de possibles, lieu d'implantation d'une nouvelle usine, parcours dans un graphe, démonstration de théorèmes.

Dans ces problèmes on a donc un espace de recherche **X fini et discret**.

Discret signifie que tous les points de X sont séparés X est en particulier discontinu. C'est sur de tels problèmes que peuvent être lancées des procédures d'énumération.

L'énumération explicite, ou algorithme du *British Museum*. Dès qu'on cherche un point appartenant à un ensemble fini on peut songer à énumérer effectivement tous les points de l'ensemble pour ne garder que celui qui convient. Cette méthode est sûre, facile à mettre en oeuvre et fait parfaitement l'affaire tant que la cardinalité de l'ensemble reste inférieure à 10^3 pour un calcul manuel ou à 10^9 pour un calcul en machine.

Le schéma en est :

- a) Prendre le premier x_0 de X non encore considéré ;
- b) Evaluer les contraintes $K(x_0)$;
- c) SI une contrainte n'est pas satisfaite, recommencer ;
- d) Sinon x_0 est une solution, (aller en 1 si on les veut toutes).

Cette énumération explicite peut en fait être aisément améliorée grâce à l'idée suivante : au lieu de construire x_0 complètement, on peut se contenter d'en préciser seulement une partie et de regarder à mesure si $K(x)$ peut être satisfait.

Ainsi, dès 1891. E. Lucas, dans ses »Récréations Mathématiques » définissait une méthode systématique pour résoudre les problèmes de labyrinthes comme celui de la figure 5.1.

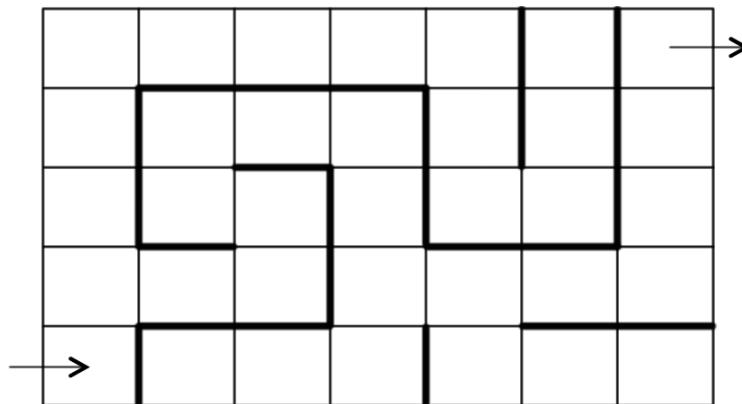


FIG. 5.1 – Parcours dans un labyrinthe

La méthode de Lucas est la suivante :

- a) A partir de la case où l'on se trouve, emprunter un chemin non encore exploré.
- b) Si tous les chemins partant de cette case ont déjà été explorés, alors revenir en arrière d'un pas en suivant le chemin qui avait été emprunté pour arriver là où l'on se trouve.

On voit qu'au lieu d'engendrer aveuglément tous les sous-ensembles de l'ensemble X des cases du labyrinthe, on ne considère que ceux qui constituent des chemins admissibles et on s'arrête dès que l'on rencontre un cul-de-sac. Cette méthode, systématisée, va donner naissance à un schéma tout à fait général de résolution des problèmes de nature combinatoire : *l'énumération implicite*. Un cas particulièrement favorable dans une énumération est celui où un gradient peut être défini.

5.2 Les méthodes de gradient

Un *gradient* est une variation (gradus = marche). Nous utilisons couramment cette notion pour résoudre des problèmes concrets : l'idée est d'avancer suivant la direction de plus grande pente pour atteindre le sommet. Quand, en voiture, nous voulons traverser une ville du nord au sud. nous nous efforçons de conserver le plus longtemps possible la direction et, si un carrefour nous force à nous éloigner, nous minimisons cet

éloignement. La stratégie du gradient ou du »*hill climbing*» (escalade) consiste à tenter d'atteindre un optimum global par une succession d'optimum locaux. L'expérience montre qu'elle est bonne, bien qu'elle soit mise en défaut dès que l'espace de recherche n'est pas convexe.

L'idée, en face d'une situation concrète quelconque, est de lui faire correspondre une *fonction d'évaluation* numérique qui prend sa valeur extrême au sommet que l'on peut atteindre, et de choisir, à chaque étape, l'action qui augmente le plus la variation de celle-ci à partir de la situation courante. Quelque peu formalisée, cette méthode a donné naissance à plusieurs familles d'algorithmes : algorithme du simplexe en programmation linéaire (chapitre 5, section 5.3), procédure A^* (chapitre 5 section 5.4), méthodes par approximations successives en analyse numérique, minimax et alpha bêta (chapitre 6 section 6.2), programmation dynamique (chapitre 5 section 5.9), « branch and bound » et algorithme par séparation et évaluation (chapitre 5 section 5.4).

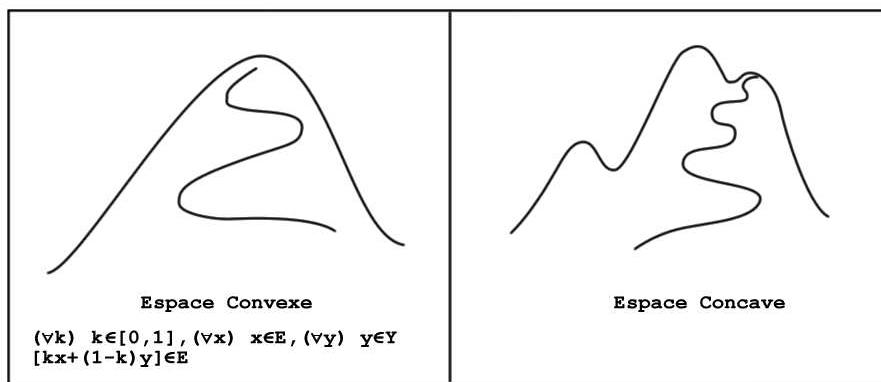


FIG. 5.2 – Méthode du gradient

Exemple 1 :

Soient cinq flèches données en position S_0 : il s'agit de les amener à la position S_b , les seuls coups permis étant tous les retournements simultanés de deux flèches adjacentes.



FIG. 5.3 – Les flèches

Les cinq actions licites a , b , c , d et e , sont donc les retournements $(1, 2)$, $(2, 3)$, $(3, 4)$, $(4, 5)$, $(5, 6)$. Comme, pour toute action α de cet ensemble, α^2 vaut l'identité, il suffit de trouver une séquence d'actions élémentaires. Si nous décidons de prendre pour fonction d'évaluation d'une situation S : $f(S) =$ le nombre de paires de flèches en bonne position, il vient $f(S_0) = f(000101) = 2$ et $f(111111) = 6$. Le coup a fait passer de S_0 à $S_1 = (110101)$ avec $f(S_1) = 4$. Le coup d fait passer de S_1 à $S_2 = (110011)$ avec $f(S_2) = 4$ et finalement (il est clair que l'ordre des coups n'intervient pas), le coup c permet d'obtenir la situation but S_b .

Nous remarquons que nous avons dû franchir un pallier, $f(S) = \text{constante}$, à la seconde

étape. Mais le parcours est très lié à la fonction d'évaluation et il n'en aurait pas été de même si nous avions retenu pour celle-ci l'expression : $f(S)$ = distance maximale entre deux flèches mal placées, puisqu'alors : $f(S_0) = 4$, $f(S_1) = 2$ et $f(S_2) = 1$.

Cette méthode ne garantit donc pas, sauf si l'espace image $f(S)$ est convexe, l'obtention de la solution, puisque trois types de pièges peuvent se présenter : les sommets secondaires, les crêtes et les plateaux.

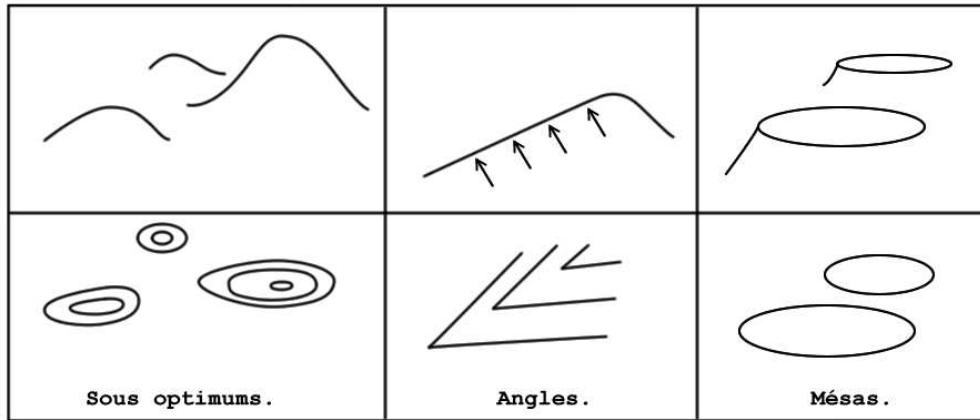


FIG. 5.4 – Echecs des méthodes de gradients

De très nombreux problèmes se situent dans le dernier cas ; Il est facile d'approcher la solution et de tout trouver sauf un détail : un immense méplat final, où l'on ne progresse plus, oblige à reconsiderer l'évaluation. Mais, pour peu que f soit bien choisie, les cas où l'espace $f(S)$ est convexe permettent d'obtenir des solutions rapides et élégantes. Les algorithmes rencontrés au paragraphe 4 section ?? étaient, par exemple, de ce type avec :

$$\begin{aligned} f(S) &= \text{distance minimale à l'origine dans les plus courts chemins;} \\ f(S) &= \text{nombre d'éléments triés dans les tris.} \end{aligned}$$

Enfin un cas particulièrement favorable et utile est celui de la programmation linéaire.

5.3 La programmation linéaire

Un programme linéaire est comme nous l'avons vu, un ensemble d'inéquations en X , de la forme :

$$A.X \leq b \quad \text{avec} \quad X \in \mathbb{R}^n$$

où A est une matrice de coefficients réels, donnée, de dimension $m \times n$, X le vecteur colonne $1 \times n$ des inconnues réelles et b le vecteur second membre réel donné $1 \times m$. En général, on a $m < n$ et le système est indéterminé. Les solutions sont alors départagées par les valeurs d'une *fonction économique* ou objectif qu'elles doivent maximiser.

Exemple :

« Une entreprise artisanale qui fabrique les meubles à façon, reçoit simultanément deux commandes. La première concerne des meubles qui demandent 11 unités de fournitures et 3 heures de fabrication par meuble fabriqué. La seconde requiert, quant à elle, 4 unités de fournitures et 5 heures de travail. L'artisan dispose de 46 unités de fournitures

et de 32 heures de travail et il sait que la première commande lui rapporte 14 francs par meuble et la seconde 7 francs seulement. Comment doit-il s'organiser ? »

Si x_1 et x_2 sont les nombres de meubles fabriqués pour chaque commande. Il vient immédiatement :

- a) $11 \times x_1 + 4 \times x_2 \leq 46$ (contrainte de fourniture) ;
- b) $3 \times x_1 + 5 \times x_2 \leq 32$ (contrainte d'horaires)
- c) $Z = 14 \times x_1 + 7 \times x_2$ maximum ;

Soit un programme mathématique du type $AX \leq b$ avec $\text{MAX}[C, X]$.

Comme il est plus simple de manipuler des égalités plutôt que des inégalités. Il est d'usage d'introduire des variables auxiliaires y_1 et y_2 , dites variables d'écart, qui permettent d'écrire le système sous la forme :

$$\begin{aligned} 11 \times x_1 + 4 \times x_2 + y_1 &= 46 \\ 3 \times x_1 + 5 \times x_2 + y_2 &= 32 \\ \text{MAX}[14 \times x_1 + 7 \times x_2] \end{aligned}$$

ou encore, en notation matricielle :

$$\text{MAX} \begin{bmatrix} 11 & 4 & 1 & 0 \\ 3 & 5 & 0 & 1 \\ 14 & 7 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 46 \\ 32 \\ 0 \end{bmatrix}$$

L'idée est de partir de la solution évidente $x_1 = 0$, $x_2 = 0$, $y_1 = 46$, $y_2 = 32$ et de l'améliorer en augmentant le plus possible la valeur de la variable qui accroît le plus la fonction économique Z .

Le »gradient», qu'on utilise est ici directement donné par la dernière ligne et, c'est visiblement la variable x_1 qui a, à la première étape, le meilleur rendement. On écrit donc :

$$x_1 = \frac{46}{11} - \frac{4}{11}x_2 - \frac{1}{11}y_1 ,$$

soit, matriciellement :

$$\begin{bmatrix} 1 & \frac{4}{11} & \frac{1}{11} & 0 \\ 0 & 5 - \frac{12}{11} & -\frac{3}{11} & 1 \\ 0 & 7 - \frac{56}{11} & -\frac{14}{11} & 0 \end{bmatrix} \cdot X = \begin{bmatrix} \frac{46}{11} \\ 32 - 3 \times \frac{46}{11} \\ -\frac{644}{11} \end{bmatrix}$$

La fonction économique Z vaut donc ainsi : $\frac{644}{11}$, lorsque x_1 est poussée à sa valeur maximum, soit $\frac{46}{11}$, tandis que x_2 vaut toujours 0 et $y_2 = 32 - 3 \times \frac{46}{11} = \frac{214}{11}$. On a, en réalité, effectué une simple translation suivant l'axe des x_1 qui montre que Z peut continuer à croître encore en augmentant cette fois x_2 , dont le gradient vaut : $7 - \frac{56}{11} = \frac{21}{11}$.

Le coefficient actuel de x_2 dans la seconde contrainte est $5 - \frac{12}{11}$, soit $\frac{43}{11}$; il vient :

$$x_2 = \frac{214}{43} + \frac{3}{43} \times y_1 - \frac{1}{43} \times y_2$$

D'où, à nouveau, matriciellement :

$$\begin{bmatrix} 1 & 0 & \frac{1}{11} + 4 \times 43 \times \frac{3}{11} & -\frac{4}{43} \\ 0 & 1 & -\frac{3}{43} & \frac{11}{43} \\ 0 & 0 & -\frac{14}{11} + 3 \times 43 \times \frac{21}{11} & \end{bmatrix} \times X = \begin{bmatrix} \frac{46}{11} - 4 \times 43 \times \frac{214}{11} \\ \frac{214}{43} \\ -14 \frac{46}{11} - 21 \times 43 \times \frac{214}{11} \end{bmatrix}$$

Cette nouvelle translation et les combinaisons linéaires qu'elle entraîne montrent que tous les gradients, tous les gains relatifs (dernière ligne de la matrice précédente) dans Z sont négatifs ou nuls. L'optimum est donc atteint. Il vaut :

$$Z^* = \text{MAX } Z = 14 \times \frac{46}{11} + 21 \times 43 \times \frac{214}{11} = 68.033$$

avec $x_1 : \frac{102}{43} = 2.371$ et $x_2 = \frac{214}{43} = 4.977$. L'espace de la recherche et l'interprétation de cet algorithme de gradient, dû à G. Dantzig (1959) et connue sous le nom de *méthode du simplexe*, peuvent être immédiatement donnés (figure 5.5), dans l'espace à deux dimensions $x_1 \times x_2$:

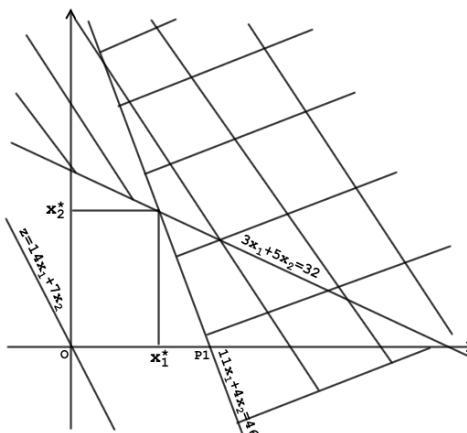


FIG. 5.5 – Méthode du simplexe en programmation linéaire

Partant du « niveau de la mer » en O avec $x_1 = x_2 = 0$, nous nous sommes successivement rendus aux points P_1 puis P_2 en suivant la **ligne de plus grande pente** selon $Z = 14 \times x_1 + 7 \times x_2$. Le « sommet » est atteint en P_2 qui est le point admissible, satisfaisant les contraintes $AX \leq b$, qui est à la plus grande distance depuis l'origine de la droite Z .

Bien sûr, la solution ne vaut qu'en nombres réels. Elle ne fournit une solution acceptable pour l'artisan que si les commandes de type 1 et 2 se répètent assez souvent pour qu'il soit raisonnable d'approcher les nombres réels de meubles trouvés par leurs arrondis entiers : $x_1 = 2$ et $x_2 = 5$. Sinon, le problème où les x_1 sont astreints à être des nombres entiers est d'une toute autre nature, puisque l'espace de recherche \mathbb{N}^n n'est même plus continu et ne saurait être convexe. C'est au chapitre 8 que seront exposées des méthodes radicalement différentes pour traiter ce dernier cas.

La méthode du simplexe s'étend sans difficulté, sous la forme matricielle, à un nombre quelconque de variables et de contraintes. La procédure se réduit à une séquence de combinaisons linéaires dont le choix est fixé par l'examen des valeurs des gradients. Elle s'arrête lorsque ceux-ci sont tous négatifs.

5.3.1 Formalisation de l'algorithme du simplexe

« L'algorithme du simplexe », dont la complexité n'est pas polynomiale, repose sur quelques propriétés simples liées aux hypothèses de travail pour les problèmes linéaires :

- a) Les inégalités : $a_1 \times x_1 + a_2 \times x_2 + \dots + a_n \times x_n \leq b$ définissent des sous-espaces fermés convexes de \mathbb{R}^n bornés par des hyperplans.
- b) Une intersection d'un nombre fini de sous-espaces convexes est un convexe appelé ici *polyèdre* s'il est borné, ou *polytope* s'il ne l'est pas.
- c) Tout point d'un polyèdre peut s'exprimer comme une combinaison linéaire des sommets ou points-extrêmes de celui-ci qui sont les intersections des hyperplans de départ. Ces points sont en nombre fini, inférieur ou égal à C_m^n dans \mathbb{R}^n , s'il y a m hyperplans (inégalités).
- d) L'optimum ne saurait être atteint en un point situé strictement à l'intérieur du polytope de définition, puisqu'alors on peut construire, par combinaison linéaire, un point sur la frontière du polyèdre où la fonction économique Z aura une valeur meilleure. Si le problème n'est pas dégénéré (si : $\text{rang}[\det A] \neq m$), la valeur de Z pourra alors encore être améliorée en suivant l'hyperarête jusqu'au prochain sommet.
- e) On appelle base de A tout sous-ensemble d'indices de $[1, m]$ auquel correspond une matrice carrée B extraite de A et de rang n . Ce qui vient d'être dit équivaut alors à la proposition : « L'optimum d'un programme linéaire $Ax \leq b$ ne peut être atteint qu'en une solution de base de A^* » : la matrice A est partitionnée en B et N et le système s'écrivant : $B \times x_B + N \times x_N = b$, une solution de base est définie de façon unique par : $x_N = 0$ et par la résolution du système de Cramer : $B \times x_B = b$. L'algorithme se contente de parcourir les bases réalisables, c'est-à-dire telles que $x_B \geq 0$, en cheminant de sommet en sommet voisin par la méthode du gradient. A chaque étape, un indice sort de la base : une variable devient nulle, et un autre entre dans la base, c'est à dire une variable prend éventuellement une valeur non nulle. La convergence est assurée par la finitude du nombre de sommets du polytope.

5.4 Méthodes de gradient en théorie des graphes

Il arrive que la méthode du gradient conduise directement et de façon certaine à la solution. Il en est ainsi dans le problème de graphe suivant, qui modélise plusieurs situations concrètes – conception de réseaux de télécommunications ou de distributions, recherche de questionnaires, classification automatique – : "Etant donné un graphe value connexe $G = (X, U)$, construire un arbre partiel $A = (X, V^*)$ de valuation totale minimale". Un arbre est un graphe $A = (X, V)$ connexe, sans cycle : Il admet donc $|X| - 1$ arêtes (formule d'Euler).

L'idée du gradient, qui conduit à considérer les arcs de G par ordre de valuations croissantes, permet d'établir le premier algorithme trouvé par Kruskal en 1956 pour résoudre ce problème.

Algorithme de l'arbre minimum

- a) Ranger les arêtes de G par ordre de valuations croissantes : $V \leftarrow \Phi$
- b) TANT QU'il existe des arêtes non considérées :
SI la première d'entre elles ne forme pas de cycle avec V
ALORS l'ajouter à V .
Fin.

Note : Cet algorithme travaille en exactement $|X|$ étapes.

5.4.1 Démonstration de l'algorithme de Kruskal

Soit L la valuation de l'arbre (X, V) obtenu à la dernière itération de l'algorithme précédent et montrons qu'il ne saurait exister d'arbre de valuation strictement inférieure à L . Soit alors, en effet, parmi ces arbres, l'arbre (X, V^*) dont la valuation L^* est la plus faible, (X, V^*) et (X, V) possèdent chacun $(|X| - 1)$ arêtes. Soit v_i , $(1 \leq i \leq n - 1)$ la première de celles-ci qui est dans V , sans être dans V^* (si elle n'existe pas le résultat est démontré). L'adjonction de v_i dans V^* crée donc un cycle. Sur ce cycle, il existe nécessairement une arête v_0 qui, elle, n'est pas dans V . Le graphe $(X, V^* + v_i - v_0)$ est alors aussi un arbre, comme comptant $(n - 1)$ arêtes et aucun cycle. Mais l'adjonction de v_0 dans $(X, v_1, v_2, \dots, v_{i-1})$ ne crée pas de cycle puisque ce graphe est un sous-arbre de V^* , c'est donc que v_0 est classée après v_i . Ainsi :

$$\text{valuation}(v_0) > \text{valuation}(v_i)$$

(on suppose ici, sans perte de généralité, que les valuations sont toutes différentes). Il en résulte finalement que la valuation de l'arbre $(X, V^* + v_1 - v_0)$, qui vaut :

$$L^* + \text{valuation}(v_i) - \text{valuation}(v_0)$$

est inférieure à la valeur supposée optimale L^* d'où une contradiction.

Ce type d'algorithme, qui prend les objets dans l'ordre où ils sont donnés par la fonction d'évaluation, a été appelée « *glouton* ». Son succès est en réalité assuré à chaque fois que le problème peut être ramené à la détermination de l'intersection de deux familles de parties indépendantes d'un même ensemble, dites *matroïdes* (Cf. Gondran et Minoux, 1979).

Exemple :

« Trouver dans le réseau EDF à haute tension français l'arbre recouvrant qui minimise la somme totale des distances selon le graphe de la figure 5.6. »

L'ordre des arêtes par distances croissantes est : (AC, FG, EF, EH, IJ, CD, GJ, AD, DG, BC, CF, FI, HI, AB, BE, FJ, FD). L'algorithme de Kruskal permet de retenir successivement : AC puis FG, EF, EH, IJ, CD, GJ ; AD est alors éliminée car elle formerait un cycle avec AC et CD déjà retenues. DG et BC viennent alors s'ajouter. On remarque que tous les sommets sont déjà atteints : l'arbre minimum est obtenu.

Une extension de cet algorithme, pour des problèmes de recherches de type général, a conduit en intelligence artificielle, aux méthodes dites de « *recherche heuristique* ».

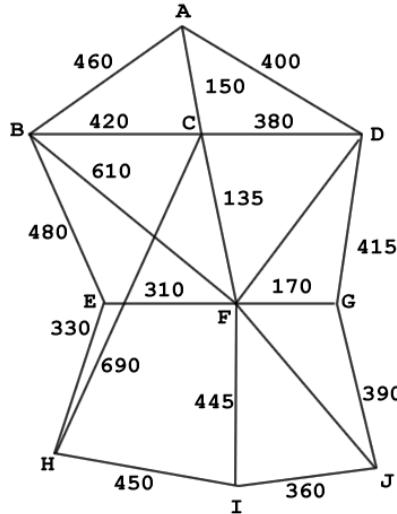


FIG. 5.6 – Graphe réseau EDF

5.5 Recherche heuristique

Une classe très générale de problèmes peut se résoudre formellement de la façon suivante :

- a) **Etape 1** : choisir une action parmi les actions possibles ;
- b) **Etape 2** : accomplir l'action retenue et modifier la situation courante ;
- c) **Etape 3** : évaluer la situation ;
- d) **Etape 4** : rejeter les situations inutiles.
- e) **Etape 5** : fin si situation finale atteinte, sinon choisir une nouvelle situation de départ et reprendre au début.

Nous avons, jusqu'ici, utilisé une seule et même fonction numérique pour les étapes de « décision » 1, 3 et 5. Nous allons maintenant affiner ce schéma. La table suivante dresse le bilan des idées et des heuristiques utilisées par l'homme et/ou par des programmes existants (table 5.1, page suivante).

Si, en suivant cette table 5.1, on décide, dans un problème de recherche d'un arbre de poids minimum :

- a) d'ajouter une arête à chaque pas dans l'arbre cherché,
- b) en 1, de choisir parmi les arêtes possibles celle de valeur minimale,
- c) en 5, de repartir toujours de l'arbre en cours de formation, on retrouve alors l'algorithme de Kruskal, démontré précédemment. Mais, on retrouve aussi des variantes de cet algorithme dont on pourra montrer qu'elles conduisent aussi à la solution ; on peut choisir d'ajouter ou d'ôter une arête à l'arbre (X, V) en construction ; en 1, il n'est plus alors utile de trier l'ensemble des arêtes ; on prend la première qui se présente ; en 3, si V contient un cycle, on ôte de X l'arête de plus forte valuation appartenant à ce cycle ; en 5, on repart de l'arbre courant (X, V) . Les autres variantes sont obtenues en tenant compte de la connectivité. Bien sûr, si ces algorithmes convergent tous, ils ne conduisent pas tous au même nombre d'opérations élémentaires. Le premier est en $O(n^3)$, le second, qui évite le tri, est en fait, de même complexité : le meilleur algorithme connu est en $O(n^2)$.

L'algorithme du simplexe entre également dans la classe des méthodes définies par la table 5.1 : une action correspond à l'entrée d'un indice dans la base et à la sortie d'un autre ; en 1, on choisit l'indice de meilleur gradient pour entrer (l'indice de sortie est alors imposé en 2 pour conserver une solution appartenant au polyèdre) ; en 5, on repart encore de la solution courante.

Dans ce type de recherche, il faut noter que dans le cas général, rien ne garantit qu'on ne rencontre pas à nouveau, en 5, une situation déjà étudiée dans un précédent contexte, ni que l'algorithme converge effectivement vers la solution quand celle-ci existe. Une réponse simultanée à ces deux questions est donnée par une version plus raffinée de la méthode de gradient générale. Elle a été proposée par Hart, Nilsson et Raphaël (1968) (1972) et est connue sous le nom *d'algorithme A**.

Etape 1 : Choix d'une action

- par pertinence vis-à-vis du but :
 - réduction d'une différence indésirable ;
 - résolution immédiate d'un sous-problème.
- par expérience :
 - répétition du passé ;
 - détection d'une action-clé.
- par condition nécessaire :
 - décision forcée par analyse de la situation ;
 - élimination par infaisabilité.
- par hasard :
 - préférence à la diversité

Etape 3 : Evaluation de la situation

- par analogie :
 - problème reconnu ;
 - sous-problème reconnu ;
- mesure d'une distance au but :
 - entre les deux situations ;
 - en fonction de l'effort estimé de recherche.
- par critère mathématique :
 - construction de propriétés nécessaires et/ou suffisantes pour la solution ;
 - fonction d'évaluation numérique ;
 - bornes supérieures et inférieures ;
 - somme de valeurs de caractéristiques satisfaisantes.
- par espérance de gain : mesure liée à l'expérience passée :
 - la simplicité de la situation ;
 - le taux de croissance de la recherche ;
 - tout autre critère (difficulté du problème, temps passé...).

Etape 5 : Choix du pas suivant

- toujours avancer :
 - développement systématique de la dernière situation engendrée.
- tout faire en parallèle :
 - prendre les actions les unes après les autres.
- repartir de la situation la plus prometteuse :
 - au sens de la fonction d'évaluation de situation ;
 - en raison de son petit nombre d'actions.
- faire un compromis entre :
 - la profondeur de la recherche ;
 - l'évaluation de la situation.

TAB. 5.1 – Méthodes de recherche heuristique

5.6 L'algorithme A^*

Chaque situation S , obtenue à partir de la situation initiale et d'une suite déterminée d'actions, reçoit une évaluation numérique $f(S)$ qui se compose de deux termes :

$$f(S) = g(S) + h(S)$$

où $g(S)$ est le coût actuel réel de la situation S et où $h(S)$ est une fonction heuristique qui donne une estimation du coût de la meilleure suite d'actions depuis S jusqu'à la solution. Au total, $f(S)$ est donc une mesure du coût des solutions « astreintes à S », c'est-à-dire comportant le même sous-ensemble d'actions initiales que S . Lorsqu'à chaque action correspond, en particulier, un coût élémentaire, $c(S_1, S_2, a)$ qui est le coût de passage de S_1 à S_2 par l'action a , la fonction $g(S)$ est très simple à calculer : c'est la somme des coûts des actions effectuées pour passer de la situation de départ S , les coûts seront dorénavant toujours supposés positifs : la fonction g est alors strictement additive.

Etape 0 : créer le graphe de recherche $R = (E, A)$ en posant :

```

 $S_0 \leftarrow E$  (situation initiale)
 $f(S_0) \leftarrow 0$  ( $E$  sera dans la suite partitionné en deux
sous-ensembles  $O$  et  $F$  de situations "ouvertes" et "fermées")
 $O \leftarrow S_0$ 
 $F \leftarrow \emptyset$ 

```

TANT QUE $O \neq \emptyset$ faire :

Etape 1 : Choisir dans O une situation S telle que $f(S)$ soit minimal :

```

 $O \leftarrow O - S$ ,  $F \leftarrow F + S$ 
Si  $S$  est solution, alors fin.

```

Etape 2 : Sinon développer la situation S par construction des situations

S_i , ($0 \leq i \leq n_s$) engendrées par les actions possibles à partir de S .

Etape 3 : Calculer $f(S_i)$ pour chaque successeur :

```

 $f(S_i) = g(S_i) + h(S_i)$ 
avec  $h(S_i) \leq h^*(S_i)$ 

```

Etape 4 : Placer dans O , avec leur évaluation, ceux des S_i

qui ne sont ni dans O , ni dans F

pour tous ceux qui appartiennent déjà à O ou à F , poser :

$f(S_i) \leftarrow \min(f(S_i)^{\text{ancien}}, f(S_i)^{\text{nouveau}})$

s'ils appartenait à F , alors en outre :

faire $F \leftarrow F - S_i$, $O \leftarrow O + S_i$

Fin tant que

Fin A^*

L'algorithme A^* est caractérisé par une propriété fondamentale sur h : si $h^*(S)$ est le coût minimal de toute suite d'actions permettant d'atteindre la solution à partir de S , alors $h(S)$ est une borne inférieure de $h^*(S)$; en outre $h(S)$ doit être positive. Ainsi :

$$0 \leq h(S) \leq h^*(S)$$

La fonction $f(S)$ est donc une évaluation de la fonction optimale $f^*(S)$ définie par :

$$f^*(S) = g^*(S) + h^*(S)$$

avec $f^*(S)$ = meilleure valeur des solutions passant par S ;
 $g^*(S)$ = meilleure valeur de S_0 jusqu'à S ;
 $h^*(S)$ = meilleure valeur de S jusqu'au but.

Théorème : L'algorithme A^* se termine s'il existe une suite finie d'actions depuis la situation initiale jusqu'à la solution.

Preuve : Une simple remarque tout d'abord : si le nombre d'actions possibles pour chaque situation est toujours fini, et que les situations sont elles-mêmes en nombre fini, alors A^* converge certainement. En effet, chaque itération ôte un sommet de l'ensemble O et en ajoute un nombre fini de nouveaux : le graphe de recherche reste fini et son examen n'exigera qu'un nombre fini d'itérations.

Supposons donc que le graphe de recherche soit infini, l'hypothèse du théorème étant cependant vérifiée. Raisonnons par l'absurde et admettons que A^* ne se termine pas. Ceci ne peut se produire que parce que O lui-même croît indéfiniment. Montrons que ceci entraînerait que la plus petite des valeurs $f(S)$ serait également infinie et donc qu'une solution à distance finie ne saurait exister : pour toute situation S , soit $v^*(S)$ le nombre minimum d'actions nécessaires pour obtenir S à partir de S_0 (la valeur $v^*(S)$ n'est pas nécessairement atteinte par A et est donc inconnue). Quoiqu'il en soit, puisque tout arc a un coût strictement positif, donc supérieur à une valeur donnée, on a pour valuation optimale, $g^*(S)$, de S :

$$g^*(S) > \epsilon \times v^*(S)$$

La véritable valeur $g(S)$ trouvée par A^* vérifie donc :

$$g(S) \geq g^*(S) > \epsilon \times v^*(S)$$

Comme la fonction heuristique $h(S)$ est elle-même positive ou nulle, il s'ensuit également que :

$$f(S) \geq g^*(S) > \epsilon \times v^*(S)$$

Ainsi, si $v^*(S)$ devient infini, il en est de même de $f(S)$ et aucune suite finie d'actions n'existe.

Mais, prouvons par ailleurs, qu'il existe nécessairement dans O , à toute étape de l'algorithme avant l'arrêt, un sommet S_2 tel que :

$$f(S_2) \leq f^*(S_0)$$

Si une séquence finie d'actions et de situations existe depuis S_0 jusqu'au but, soit alors S_2 la première des situations de cette séquence qui soit aussi dans O . Un tel S_2 existe tant que A^* n'est pas terminé. Comme pour toute situation : $f(S_2) = g(S_2) + h(S_2)$; mais ici, de plus : $g(S_2) = g^*(S_2)$ car toutes les situations antérieures sont dans F , par définition de S_2 . En outre, par construction de A^* , l'inégalité $h(S_2) \leq h^*(S_2)$ est vérifiée. Il vient finalement :

$$f(S) \leq g^*(S_2) + h^*(S_2), \text{ soit } f(S_2) \leq f^*(S_2)$$

En particulier, si la séquence de départ était optimale, alors :

$$f(S_2) \leq f^*(S_0)$$

L'hypothèse de non finitude de f et donc celle de A^* est ainsi contredite, même pour des graphes localement infinis, sous réserve de l'existence d'une solution à distance finie.

La solution trouvée est, de plus, de coût optimal. Si l'arrêt de A^* se produisait en effet, en un noeud S_3 tel que : $f(S_3) > f^*(S_0)$, alors la preuve ci-dessus montre qu'à l'étape précédente, il existerait un noeud S_2 tel que : $f(S_2) \leq f^*(S_0)$, c'est S_2 et non S_3 qui aurait été sélectionné à l'étape 1. En fait, tout noeud S développé par A^* possède une évaluation $f(S)$ au plus égale à $f^*(S_0)$ puisque sinon, S est soit un noeud solution, soit n'est pas choisi.

Bien sûr, la fonction f doit être la plus proche possible de la quantité évaluée, $f^*(S_0)$, tout en lui restant inférieure. Plus grande sera la valeur de $h(S)$, plus efficace sera l'algorithme. On montre facilement en effet que si : $h_2(S) > h_1(S)$, tout sommet développé par la fonction h_2 le sera par la fonction h_1 . Si la fonction h vérifie la propriété de monotonie vis-à-vis des coûts locaux, c'est-à-dire si :

$$h(S_2) \leq h(S_1) + C(S_1, S_2, a)$$

on montre encore qu'alors, pour tous les sommets développés, A^* a déjà construit une séquence optimale jusqu'à eux :

$$S \text{ développé} \Leftrightarrow g(S) = g^*(S)$$

La suite des valeurs $f(S)$ des sommets développés ne peut alors être que non-décroissante, assurant ainsi une convergence rapide. Ce dernier cas n'est d'ailleurs qu'une variante de l'algorithme de Moore-Dijkstra (1957) pour la recherche du plus court chemin dans un graphe (Cf. paragraphe 4.2). Le gros défaut de l'algorithme est bien entendu le cas où il n'existe pas de solution au problème posé...

Exemple d'application de l'algorithme A^* : Nous reprenons ici, après N. Nilsson, un casse-tête célèbre inventé par le génial Sam Loyd : le taquin. Passer dans un taquin à 9 cases, de la configuration :

2	8	3
1	6	4
7	□	5

à la configuration

$S_{but} =$

1	2	3
8	□	4
7	6	5

(Le symbole \square désigne une case vide : toute case voisine horizontalement ou verticalement peut venir s'y placer en un coup). On veut minimiser le nombre de coups. Retenons pour fonctions d'évaluation :

$$\begin{aligned} g(S) &= \text{nombre de coups de } S_0 \text{ jusqu'à } S, \\ h(S) &= \text{nombre de cases mal placées.} \end{aligned}$$

On a bien ainsi :

- a) $g(S) \leq g^*(S)$ par construction de g ;
- b) $h(S) \leq h^*(S)$ par la nature du problème.

Par contre rien ne garantit la monotonie de h . La figure 5.7 donne l'arbre de recherche développé par A^* avec $f(S) = g(S) + h(S)$, et à droite de chaque situation, le couple de valeur $g(S), h(S)$ et la numérotation née par l'algorithme.

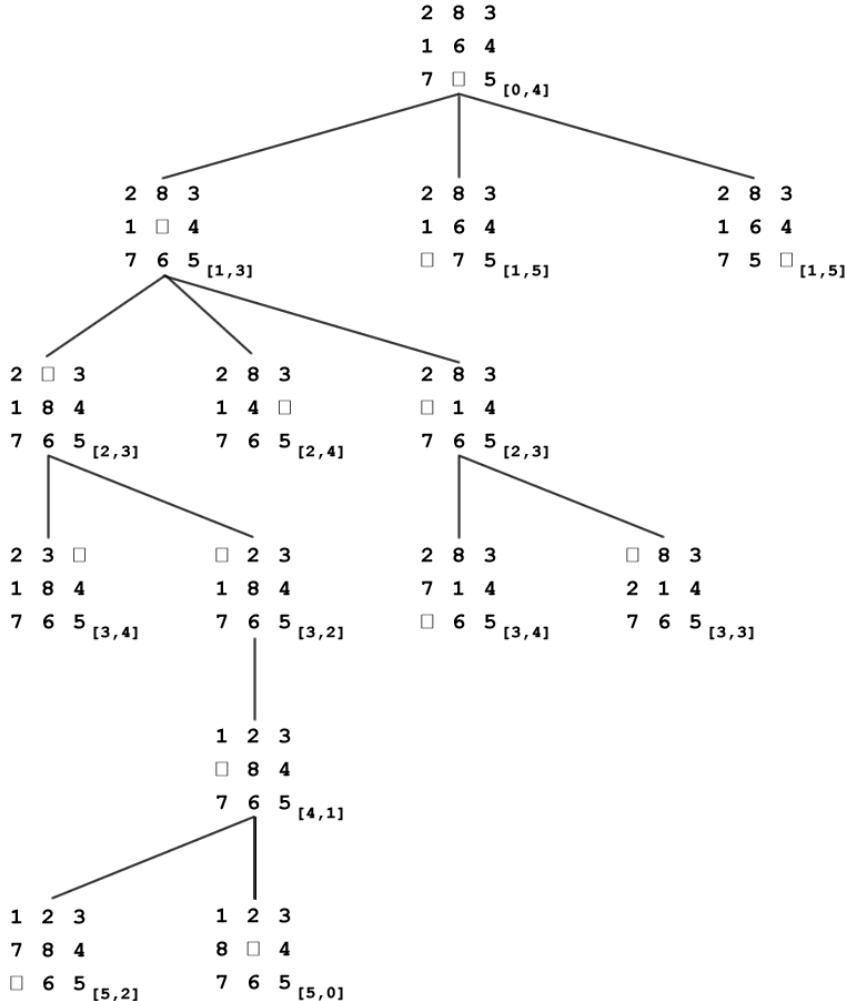


FIG. 5.7 – Recherche A^* pour un taquin

Des variantes de cet algorithme ont été proposées. Une forme plus générale de la fonction $f(S)$ est ainsi définie par I. Pohl (1970) :

$$f(S) = (1 - \alpha)g(S) + \alpha \times h(S)$$

où α est une constante réelle choisie entre 0 et 1. Une pondération est ainsi effectuée dans l'évaluation entre la partie rigoureuse $g(S)$ et la partie heuristique $h(S)$. L'algorithme A^* correspond à la valeur $\alpha = \frac{1}{2}$. Le cas $\alpha = 1$ correspond à l'idée du gradient pur. L'autre cas extrême $\alpha = 0$ donne au contraire une méthode d'énumération complète que nous avons rencontré sous le nom d'algorithme du « British Museum ». Rien n'oblige en réalité le coefficient α à être constant pendant toute la recherche ; il est

ainsi possible de faire décroître sa valeur à mesure que l'on progresse et que l'évaluation heuristique perd donc de son importance : les meilleures valeurs de α peuvent même être apprises a posteriori, pour un problème donné, par un programme de façon à minimiser le nombre de sommets développés. Enfin, la recherche peut être bidirectionnelle : on part à la fois en avant de la situation initiale, mais aussi en arrière par inversion des actions possibles à partir du but, la recherche s'arrêtant lorsqu'une même situation est rencontrée dans les deux graphes de recherche.

Analyse critique :

L'algorithme A^* , comme tous les autres algorithmes de gradient présentés dans ce chapitre, est un algorithme strictement numérique : en tant que tel il exclut l'analyse formelle de chaque situation ; ainsi, A^* repose sur l'évaluation du coût d'une solution. Mais il existe maints problèmes où cette notion n'a pas de sens, parce que seule une solution réalisable par exemple est recherchée : il n'est, en tant que tel, d'aucune aide pour calculer une intégrale, résoudre un système d'équations ou faire un diagnostic médical. Les impasses, les bouclages ne sont pas détectés à l'avance : si dans le problème du taquin, la rangée du haut dans la situation initiale est changée en (8 2 3), A^* explorera absolument tous les cas possibles pour montrer que le problème n'a plus de solution ; il n'est plus d'aucune utilité ; d'autres arguments (chapitre 8) peuvent pourtant parvenir d'emblée au résultat. Ce type d'algorithme ne doit donc être réservé qu'aux problèmes posés dans des univers mal connus, où peu d'informations sont disponibles : faible connaissance du contexte, manque d'expérience, absence d'arguments formels.

Un problème de pesées : « *On dispose d'une balance de Roberval et on veut trouver dans un ensemble de douze boules celle qui ne pèse pas le même poids que les onze autres (on ne sait pas si elle existe effectivement ni si elle est plus lourde ou plus légère). Le nombre de pesées doit être minimal* ».

Ce problème est assez connu, mais nous nous intéressons ici surtout à la façon de démarrer la résolution. Nous pouvons, bien sûr, comparer les boules deux à deux sur la balance. Comme une boule, au plus, est différente des autres, nous sommes assurés de parvenir à la réponse en au plus : 11 (nombre de comparaisons de la boule 1 avec les autres) + 1 (détermination du sens de la différence), soit 12 pesées au total. Si nous décidons de commencer au contraire par une pesée à 6 contre 6, nous n'apprendrons évidemment rien. Dans ces conditions, la première question qui se pose est celle de la détermination du nombre de boules à placer sur chaque plateau de la balance la première fois (seules les pesées équitables sont considérées, les autres ne pouvant apporter aucune information).

Une fonction d'évaluation et la mesure de son gradient peuvent nous donner la réponse. Chaque pesée doit faire croître l'information que nous possédons sur l'ensemble des boules. Il s'agit donc de rendre *l'espérance mathématique* du nombre de boules inconnues aussi petite que possible (il s'agit bien d'espérance, puisque jusqu'à l'étape ultime nous ne sommes sûrs de rien). Or, si nous démarrons avec b boules ($2b \leq n$) sur chaque plateau, ou bien le fléau marque l'égalité et alors $(12 - 2b)$ boules restent de poids inconnus : ou bien le fléau s'incline d'un côté et alors $(12 - 2b)$ boules sont nécessairement de poids standard, tandis que b boules sont peut-être plus lourdes et b autres boules peut-être plus légères : ces deux cas s'excluent bien sûr mutuellement. Tout ce passe donc ici comme si b boules étaient de poids inconnus.

Notre fonction d'évaluation, $f(b)$, qui mesure l'intérêt de commencer les pesées avec $2b$ boules, est donc naturellement prise égale à ce nombre de boules inconnues dans

chaque cas, multiplié par la probabilité de rencontrer ce cas. Puisqu'il existe C_m^k façons de choisir k objets parmi m , la probabilité de rencontrer l'égalité est :

$$\frac{C_{11}^{2b}}{C_{12}^{2b}}$$

car, alors, la boule irrégulière a été laissée de côté. Le choix s'effectue sur les 11 autres boules : la probabilité de voir le fléau pencher d'un côté ou de l'autre est le complément à 1 de la précédente ; comme on a $C_{m-1}^k = \frac{(m-k)}{m} C_m^k$, notre évaluation $f(b)$ vaut :

$$f(b) = \frac{12 - 2b}{12} \times (12 - 2b) + \left(1 - \frac{12 - 2b}{12}\right) \times b$$

$$\text{soit : } f(b) = \frac{1}{12} ((12 - 2b)^2 + 2 \times b^2)$$

Le minimum de cette fonction sera obtenu en prenant sa dérivée, en effet $f(\pm\infty) = \infty$; donc b doit vérifier : $-2 \times 2 \times (12 - 2b) + 4b = 0$. soit : $b = 4$ (et d'une façon générale $b = \frac{n}{3}$ s'il y a n boules).

Bien sûr, cette valeur n'est qu'une indication du meilleur chemin à suivre qui n'avait rien d'intuitif. Elle conduit alors aisément à une solution en trois pesées. Il est même remarquable qu'on puisse définir ces pesées a priori indépendamment des résultats pour chacune d'elle. Nommant les boules de A jusqu'à L, on obtient par exemple le schéma :

$$\begin{aligned} 1 &\rightarrow A B C D \text{ contre } E F G H \\ 2 &\rightarrow A B C E \text{ contre } D I J K \\ 3 &\rightarrow B E G I \text{ contre } C H K L \end{aligned}$$

qui permet bien de distinguer les $12 \times 2 = 24$ possibilités.

Il est en fait facile de montrer que 3 est le nombre minimal de pesées puisque, justement, chaque pesée présente trois issues possibles. Deux pesées permettent de distinguer $3 \times 3 = 9$, ce qui ne saurait suffire. La méthode du gradient nous a donc ici conduit à l'optimum.

5.7 L'énumération implicite par propagation de contraintes

Illustrons la méthode de Lucas sur un problème classique mais d'énoncé assez différent du problème de labyrinthe : « *Placer sur l'échiquier de côté n le plus grand nombre possible de dames sans que deux dames quelconques soient en prise* ».

Etant donné que toute dame prend sur sa ligne, sur sa colonne et sur ses deux diagonales, il est clair qu'il est possible de disposer au plus n dames sur l'échiquier $n \times n$. Mais cela est-il réalisable ?

Première idée : considérons les lignes une à une.

Cas de l'échiquier 4×4 : nous plaçons la première dame sur la première case de la première ligne (fig. 5.8).

Alors il est évident qu'avec cette hypothèse et avec les contraintes imposées aucune solution ne peut comporter une dame en première colonne sur la deuxième ligne :

	C1	C2	C3	C4
L1	□			
L2				
L3				
L4				

FIG. 5.8 – Le problème des dames, sur l'échiquier

une des contraintes $K(x)$ serait insatisfaite, il en va de même de la seconde colonne, deuxième ligne. Le premier choix qui ne mène pas directement à un cul-de-sac pour la seconde ligne est donc le placement en troisième colonne (fig. 5.9).

	C1	C2	C3	C4
L1	□			
L2			□	
L3				
L4				

FIG. 5.9 – Second choix

Après ces deux premières hypothèses, ni les colonnes 1 et 3 à cause de la première dame, ni les colonnes 2 et 4 à cause de la seconde ne conviennent pour placer la troisième dame. Cette solution partielle, qui ne compte pourtant que deux dames, ne peut ainsi en aucun cas faire partie d'une solution complète, puisqu'aucune place n'est disponible pour une troisième dame. Il convient ainsi de remettre en cause la dernière hypothèse. Nous plaçons donc la deuxième dame en quatrième colonne (fig. 5.10).

	C1	C2	C3	C4
L1	□			
L2				□
L3				
L4				

FIG. 5.10 – Deuxième second choix dans le problème des quatre dames

Placer la troisième dame en colonne 1 viole alors $K(x)$, nous la plaçons donc en colonne 2. Placer la quatrième dame en colonne 1, 2, 3 ou 4 est en contradiction avec l'ensemble des contraintes données. Il faut donc remettre une nouvelle fois en cause la dernière hypothèse. Cette hypothèse concernait le placement de la seconde dame. Mais il ne reste plus aucune colonne disponible pour celle-ci : c'est donc en fait sur le placement de la première dame qu'il convient de revenir. Plaçons cette dame de la première ligne en deuxième colonne. Les première, deuxième et troisième colonnes conduisent à des impasses en seconde ligne. Nous plaçons donc la deuxième dame en quatrième colonne (fig. 5.11).

	C1	C2	C3	C4
L1		□		
L2				□
L3				
L4				

FIG. 5.11 – Les deux nouveaux premiers choix

La première case convient en troisième ligne et à la suite des impossibilités en colonnes 1 et 2, il ne reste qu'une place pour la quatrième dame en troisième colonne (fig. 5.12).

	C1	C2	C3	C4
L1		□		
L2				□
L3		□		
L4				□

FIG. 5.12 – La solution du problème des quatre dames

Nous avons donc obtenu une solution à notre problème. En fait, nous les avons toutes. En effet, nous avons envisagé toutes les solutions dans lesquelles la première dame était en colonne 1 ou en colonne 2. Or les autres solutions, qui correspondent aux placements de cette dame en colonne 3 et 4, se déduisent des premières par symétrie par rapport à l'axe vertical qui coupe l'échiquier en son milieu. Il y a donc au total seulement deux solutions à notre problème sur l'échiquier 4×4 .

Afin d'être à même d'étendre la méthode utilisée à d'autres problèmes, introduisons maintenant une notation. Pour chaque ligne, soit pour chaque niveau d'hypothèses, nous avons a priori n choix possibles. Chacun d'eux correspond à l'une des colonnes de notre échiquier (ici $n = 4$).

Ainsi, nous repérons toute solution partielle comprenant k dames placées sur les k premières lignes, par un vecteur v :

$$v = (y_1, y_2, \dots, y_k)$$

où y_i désigne le numéro de la colonne sur laquelle est placée la i ème dame. La recherche que nous avons menée consiste en fait à parcourir l'arbre des différentes possibilités suivant le schéma de la figure 5.13 :

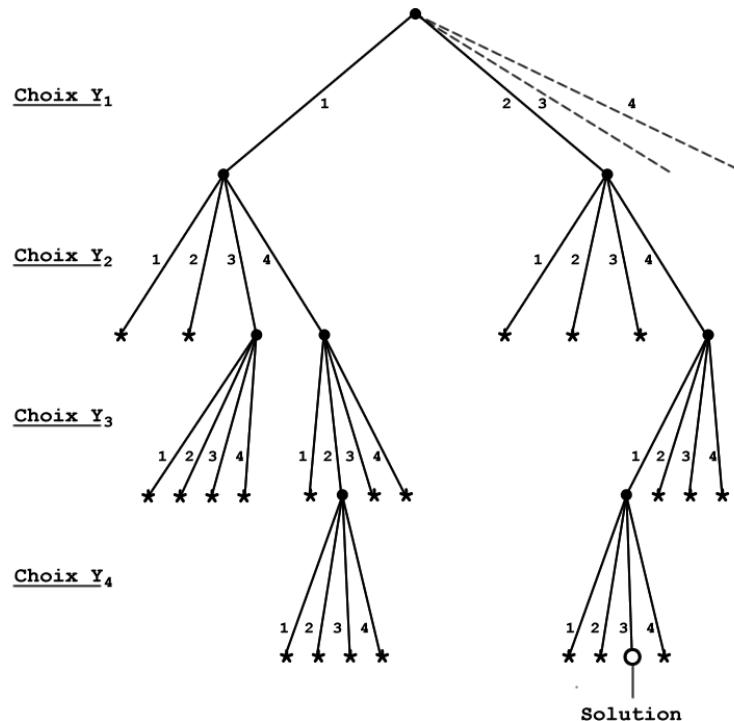


FIG. 5.13 – Arbre de recherche en énumération implicite.

Cet « arbre de recherche » a été parcouru en profondeur d'abord. Nous sommes en effet descendus le plus bas possible, tant qu'aucune impasse n'était rencontrée, et de gauche à droite, de façon systématique. Nous remarquons qu'en outre, ce même arbre peut être utilisé aussi bien pour obtenir une seule solution (on arrête le parcours quand celle-ci est obtenue) que toutes les solutions.

A chaque fois que l'espace de recherche X aura la structure de notre échiquier, nous pourrons avoir recours, pour résoudre le problème $[x?, x \in X, K(x)]$ à une recherche suivant un arbre de ce type. Plus précisément la structure particulière de l'échiquier dont nous nous sommes servis et qui justifiait à la fois la notation vectorielle et cette recherche est celle de produit cartésien. Ainsi l'échiquier a-t-il été précédemment considéré comme l'union de n lignes identiques, si bien que l'espace X n'était autre que le produit :

$$X = Y_1 \times Y_2 \times \dots \times Y_n$$

dans lequel Y_i désigne la i ème ligne de l'échiquier, qui comporte elle-même n cases. L'énumération implicite revient alors à exécuter le schéma suivant :

Algorithme général d'énumération explicite : ENUM.

Enum

(Principe : on énumère tous les éléments d'un ensemble X .
 X est un produit cartésien d'ensembles Y_i , $i \in [1, n]$.
Les ensembles Y_i sont pris dans l'ordre naturel de 1 à n .)

$i \leftarrow 1$; Calculer Y_i .

REPETER TANT QUE $i \geq 1$ Parcours de X

REPETER TANT QUE $Y_i \neq \text{vide}$ Parcours de Y_i

 Choisir un élément y_i de Y_i

$Y_i \leftarrow Y_i - \{y_i\}$

 (*on ôte ici l'élément choisi y_i de Y_i , afin de ne plus jamais reconSIDérer la même situation.*)

 SI $i < n$

 ALORS Calculer Y_{i+1} en fonction de $K(x)$

$i \leftarrow i + 1$ (*On avance dans X*)

 SINON Imprimer solution y_1, y_2, \dots, y_n

 FSI

 FR sur Y_i

$i \leftarrow i - 1$ Retour arrière

 FR sur i

FIN Enum

Dans le cas du problème des huit dames ce schéma devient :

5.7.1 Huit dames, première version.

L'échiquier est un produit cartésien de 8 lignes Y_i ($i = 1$ à 8). La recherche s'effectue par i croissant de 1 à 8. A tout instant, pour toute ligne, $\text{col}(i)$ désigne l'élément courant de Y_i c'est-à-dire la première colonne non encore essayée sur la ligne i .

```

(colonne non encore essayée sur la ligne i)
i  $\leftarrow$  1 ; (indice de ligne)
col(i)  $\leftarrow$  1 ;
REPETER TANT QUE  $i \geq 1$ 
    Essai : REPETER TANT QUE [col(i)  $\geq 8$ ]
        col(i)  $\leftarrow$  col(i) + 1
        Test de faisabilité avec les placements précédents :
            REPETER POUR ip  $\in [1 : i - 1]$ 
                SI dame(i, col(i)) en-prise-avec dame(ip, col(ip))
                    ALORS ALLER EN Essai
                    (échec partiel : essai de la colonne suivante
                    FSI
                    FR POUR ip
                    SI i  $< 8$ 
                        ALORS col(i + 1)  $\leftarrow 0$  (succès partiel)
                        i  $\leftarrow$  i + 1
                        SINON Imprimer solution col(i) (succès complet)
                        FSI
                    FR SUR col
                    i  $\leftarrow$  i - 1 (Retour arrière)
                    FR SUR i
FIN 8-dames

```

Nous allons maintenant améliorer ce schéma initial de deux manières successives :

5.7.2 Implications liées à un choix

Parmi les propriétés $K_j(x)$, avec $j = 1, 2, \dots, m$, figurant dans $K(x)$, il se peut qu'il en existe qui soient suffisamment simples, i.e. qui contiennent peu d'inconnues, pour que la connaissance des premières coordonnées y_1, y_2, \dots, y_k permette déjà de déduire des informations précises nécessaires pour que $K(x)$ soit satisfait une fois les valeurs : (y_1, y_2, \dots, y_k) fixées. Ces informations concerneront les différentes valeurs possibles pour les inconnues qui n'ont pas encore reçu de valeur, soit les y_i avec $i \geq k$.

Ainsi dans le problème de l'échiquier 4×4 le fait de poser $y_1 = 1$ implique $y_2 \geq 3$, ainsi d'ailleurs que $y_3 \neq 1$ et $y_3 \neq 3$, $y_4 \neq 1$ et $y_4 \neq 4$. Puis l'essai $y_2 = 3$ implique à son tour $y_3 \neq 2$ et $y_3 \neq 4$, c'est-à-dire un cul-de-sac, puisqu'au total aucune valeur n'est disponible pour la quatrième dame. Ceci conduit à la deuxième idée pour améliorer l'énumération.

Deuxième idée :

Afin d'éviter des tests redondants, il faut tenir compte de ces implications à mesure et donc mettre à jour après chaque hypothèse les valeurs possibles à la k ième variable y_k dans son ensemble de définition Y_k . Mais en outre, il faut pouvoir revenir en arrière. Une façon très commode de procéder est de faire correspondre à chaque ensemble Y_k un vecteur où seront comptabilisées à la fois les valeurs possibles et les valeurs interdites, avec, pour celles-ci, le numéro choix qui est cause de l'interdiction. Soit L_k ce vecteur qui enregistre les « *libertés* » pour l'ensemble Y_k . Au début, on a, pour tout k : $L_k \leftarrow 0$. Ensuite : $L_k(j) = 0$ signifie que j est une valeur encore possible pour y_k sinon : $L_k(j) = i$, où i est positif, désigne le premier choix qui provoqua l'interdiction de la valeur j pour Y_k . Ainsi dans le problème des dames sur l'échiquier 4×4 , il vient

successivement :

$$k = 1$$

$$L_1 = L_2 = L_3 = L_4 = (0, 0, 0, 0)$$

$$y_1 = 1$$

Choix 1 : $L_1 = (1, 1, 1, 1)$

$$L_2 = (1, 1, 0, 0)$$

$$L_3 = (1, 0, 1, 0)$$

$$L_4 = (1, 0, 0, 1)$$

alors : $y_2 = 3$ (directement puisque 1 et 2 sont interdits par L_2)

Choix 2 : $L_2 = (1, 1, 2, 2)$

$$L_3 = (1, 2, 1, 2)$$

$$L_4 = (1, 0, 2, 1)$$

A ce stade, le vecteur L indique qu'il n'y a plus de possibilité pour y_3 . Il faut faire machine arrière ; ceci s'effectue sans difficulté d'une part en revenant sur le choix 2 par $L_2 = (1, 1, 1, 0)$, d'autre part, en effaçant les implications liées à ce choix :

$$L_3 = (1, 0, 1, 0) \text{ et } L_4 = (1, 0, 0, 1)$$

Notons bien que : $L_2 = (1, 1, 1, 0)$ car tout se passe, on le sait maintenant, comme si la valeur 3 était interdite à y_2 en raison du seul choix 1. Tout ceci est très facile à gérer, vérifiez-le sur l'échiquier, en associant aux interdictions, par exemple, des jetons numérotés. Les vecteurs L sont des lignes de cases de l'échiquier, porteuses de jetons pour les cases interdites, vides dans le cas contraire. Le schéma de contrôle de notre méthode d'énumération implicite est ainsi devenu :

5.7.3 Huit dames, version 2.

Principe : par rapport à la version 1, l'idée essentielle est de changer les tests de faisabilité a posteriori, en implications propagées a priori après chaque choix : toutes les « cases-en-prise » encore libres deviennent interdites. Elles sont marquées, dans le vecteur libertés, par l'indice de ce choix (cf figure 5.15).

On obtient alors, pour notre problème, une façon de faire qui est très proche de la méthode du labyrinthe de Lucas :

L1	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1
1	1	1	1		
L2	<table border="1"><tr><td>1</td><td>1</td><td>2</td><td>2</td></tr></table>	1	1	2	2
1	1	2	2		
L3	<table border="1"><tr><td>1</td><td>2</td><td>1</td><td>2</td></tr></table>	1	2	1	2
1	2	1	2		
L4	<table border="1"><tr><td>1</td><td></td><td>2</td><td>1</td></tr></table>	1		2	1
1		2	1		

L1	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1
1	1	1	1		
L2	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>2</td></tr></table>	1	1	1	2
1	1	1	2		
L3	<table border="1"><tr><td>1</td><td>3</td><td>1</td><td>2</td></tr></table>	1	3	1	2
1	3	1	2		
L4	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>1</td></tr></table>	1	2	3	1
1	2	3	1		

L1	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1
1	1	1	1		
L2	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>2</td></tr></table>	1	1	1	2
1	1	1	2		
L3	<table border="1"><tr><td>3</td><td>1</td><td>2</td><td>1</td></tr></table>	3	1	2	1
3	1	2	1		
L4	<table border="1"><tr><td>3</td><td>1</td><td>4</td><td>2</td></tr></table>	3	1	4	2
3	1	4	2		

FIG. 5.14 – La solution du problème des quatre dames.

```

 $\forall u \in [1, 8], \forall v \in [1, 8], libert(u, v) \leftarrow 0$  (toutes les libertés)
 $i \leftarrow 1$ 
REPETER TANT QUE  $i \geq 1$ 
    REPETER TANT QUE  $col(i) \leq 8$ 
         $col(i)$  est le premier élément de  $E_i$  tel que  $libert(i, col(i)) = 0$ 
         $col(i) \leftarrow$  premier zero de  $libert(i, j)$ ,  $col(i) = 9$  s'il n'y en a pas
         $j \leftarrow col(i)$ 
        (le teste de faisabilité est remplacé par des interdictions effectives)
         $libert(i, j) \leftarrow i$ 
    REPETER POUR  $k \in [i + 1, 8]$ 
        REPETER POUR  $l \in [1, 8]$ 
            SI  $libert(k, l) = 0$  ALORS
                SI  $case(k, l)$  en-prise-avec  $case(i, j)$  ALORS  $libert(k, l) \leftarrow i$ 
            FR SUR  $l$ 
        FR sur  $k$ 
        SI  $i < 8$ 
    FR SUR  $col$  (retour arrière)
    REPETER POUR  $k \in [i + 1, 8]$  (Restaurer les libertés)
        REPETER POUR  $l \in [1, 8]$ 
            SI  $libert(k, l) = 0$  ALORS
                SI  $libert(k, l) = i$  ALORS  $libert(k, l) \leftarrow 0$ 
            FSI
        FR SUR  $l$ 
    FR sur  $k$ 
     $libert(i, col(i)) \leftarrow i - 1$ 
     $i \leftarrow i - 1$ 
FR SUR  $i$ 

```

FIG. 5.15 – Huit dames, version 2

5.8 Choix de l'ordre des choix

Après cette première modification, il saute aux yeux que l'ordre retenu pour les choix, qui prend successivement en compte les ensembles Y_1, Y_2, \dots, Y_n dans l'ordre naturel, n'a aucune raison d'être optimal. A l'évidence, il peut même se faire qu'un certain Y_i avec $i > k$ n'offre plus aucun élément valide alors même que, par le schéma précédent, on poursuivra l'énumération plus haut, au niveau k . La première chose à faire est, dès lors, de ne pas appeler choix ce qui n'en est pas : parmi les implications qui étaient jusqu'à présent des interdictions, il y a des implications particulières qui sont les affectations forcées de variables, du type :

S'il n'y a qu'un zéro dans le vecteur L_i , alors il faut faire $y_i = \text{indice de ce zéro}$, ceci quel que soit i et en tirer de nouvelles implications. En outre, cette amélioration a un corollaire : si le « degré de liberté » de Y_k , c'est-à-dire le nombre de 0 dans L_k , est non plus un mais deux, l'ensemble Y_k est un bon candidat pour continuer la recherche. Plus généralement, on a donc pour notre approche énumérative, la troisième idée.

Troisième idée :

Faire les choix les plus contraints. Il s'agit bien de considérer à chaque étape, l'ensemble Y_i dans lequel il reste le plus petit nombre de disponibilités. Le schéma final devient celui de la version 3.

Les libertés pour chaque ensemble Y_i sont, comme auparavant, enregistrées dans les vecteurs L_i . L'ordre dans lequel les choix sont effectués n'est plus par définition l'ordre « naturel » : $1, 2, \dots, n$: le vecteur Choix sert donc à retenir l'ordre réel. L'élément particulier y_k qui a été pris dans chaque Y_k est lui-même enregistré dans le vecteur V . En outre, nous tenons ici compte des affectations forcées, c'est-à-dire de toutes les situations où tous les y sont interdits sauf un, pour un k donné. Un choix unique peut ainsi entraîner en cascade plusieurs affectations. Toutes ces affectations sont alors mémorisées dans une pile de nom Pile, qui est un vecteur, vide au départ, dans lequel on range, au fur et à mesure, l'indice de tout ensemble Y_k pour lequel une affectation est retenue. Le niveau de cette pile, c'est-à-dire le nombre d'éléments qu'elle contient, est repéré par la variable m . Enfin, lors d'un retour arrière, les libertés sont restaurées et la pile vidée jusqu'au niveau où elle était avant le dernier choix.

Huit dames : Version 3

Principe : par rapport à la version 2, l'idée principale est de ne plus considérer les lignes de l'échiquier dans l'ordre « naturel », mais de s'intéresser, à chaque étape, à la ligne la plus contrainte c'est-à-dire à celle possédant le plus petit nombre de libertés.

Ce dernier algorithme permet d'améliorer à ce point les performances que le cas de l'échiquier habituel peut être traité en entier à la main en quelques heures. Montrons la rapidité avec laquelle l'espace de recherche est examiné par cette méthode. Remarquons d'abord qu'il n'y a plus aucune raison de commencer par faire des hypothèses sur la première ligne pour la première dame. Débutons donc par la quatrième ligne par exemple. Supposons que nous en soyons arrivés, à une certaine étape de la recherche, à considérer le placement de la première dame en colonne c sur la ligne IV , et celui de la seconde en colonne f sur la ligne III : $y = (., ., f, c, ., ., ., .)$. L'échiquier sur lequel sont enregistrées interdictions et implications se présente selon la figure 5.16.

<pre> $n \leftarrow 0$ (<i>nombre de choix effectués</i>) $m \leftarrow 0$ (<i>nombre de dames placées</i>) $\forall i \in [1, 8], col(i) \leftarrow 0$ (<i>colonne choisie en ligne i</i>) $\forall j \in [1, 8], libert(i, j) \leftarrow 0$ (<i>mise à zéro des libertés</i>) $i \leftarrow 1$ REPETER TANT QUE $i \geq 1$ $ii \leftarrow$ indice de la ligne dont le vecteur liberté contient le moins de zéros (soit $nz(ii)$ ce nombre de zéros) SI $nz(ii) \geq 1$ ALORS SI $nz(ii) \neq 1$ ALORS $i \leftarrow i + 1$, $choix(i) \leftarrow ii$ FSI $m \leftarrow m + 1$ $col(ii) \leftarrow$ premier-zéro-de $libert(ii, j)$ (<i>éventuellement unique</i>) $libert(ii, col(ii)) \leftarrow i$ (<i>implications liées à ce choix</i>) REPETER POUR $k \in [i + 1, 8]$ SI $col(k) = 0$ ALORS REPETER POUR $l \in [1, 8]$ SI $libert(k, l) = 0$ ALORS SI $case(k, l)$ en-prise-avec $case(i, j)$ ALORS $libert(k, l) \leftarrow i$ (on marque avec le dernier choix) FSI FSI FR SUR l FSI FR SUR k SI $m = 8$ ALORS Imprimer-solution $col(i)$; FSI SINON (<i>Retour arrière : aucun zéro sur une ligne au moins</i>) REPETER POUR $k \in [i + 1, 8]$ SI $col(k) = 0$ ALORS REPETER POUR $l \in [1, 8]$ SI $libert(k, l) = i$ ALORS $libert(k, l) \leftarrow 0$; FSI FR SUR l FSI FR SUR k $libert(i, col(i)) \leftarrow i - 1$; $i \leftarrow i - 1$ FSI FR SUR i FIN huit dames version 3 </pre>

	a	b	c	d	e	f	g	h
I			1	2		1		2
II	1		1		1	2	2	
III	2	1	1	1	2	2	2	2
IV	1	1	1	1	1	1	1	1
V		1	1	1		2		2
VI	1		1		1	2		
VII		2	1			1		
VIII	2		1			2	1	

FIG. 5.16 – Les huit dames

Les valeurs des degrés de libertés pour les lignes restantes sont respectivement : 4, 3 puis 3, 4, 5 et enfin 4. Pour les colonnes les degrés sont : (3, 4, *, 4, 4, *, 4, 4). C'est

ainsi la ligne V qui est une des trois rangées les plus contraintes : le troisième choix la concerne. Décidant de placer la troisième dame en colonne a , première place disponible sur cette ligne, nous obtenons l'échiquier de la figure 5.17.

	a	b	c	d	e	f	g	h
I	3		1	2	3	1		2
II	1		1	3	1	2	2	
III	2	1	1	1	2	[2]	2	2
IV	1	1	[1]	1	1	1	1	1
V	[3]	1	1	1	3	2	3	2
VI	1	3	1		1	2		
VII	3	2	1			1		
VIII	2		1	3		2	1	

FIG. 5.17 – Les huit dames (suite)

Le choix de la quatrième dame pourrait se faire, sur la seconde ligne par exemple, qui ne compte que deux libertés. Mais, à bien prendre en compte les implications, on en vient à considérer aussi les colonnes : en colonne d et e , seules deux places sont disponibles également. Or, comme pour les lignes, une dame exactement doit finalement se trouver en chaque colonne. On voit ainsi, sur ce problème particulier des huit dames, comment les contraintes de $K(x)$ peuvent agir indirectement et permettre des hypothèses judicieuses.

Ainsi l'une des cases libres en colonne d , la case (VII, d) provoquerait un retour immédiat en arrière car elle interdit simultanément les deux cases restantes sur la colonne e . Le placement de la quatrième dame en (VI, d) dans cette position est donc obligé. Bloquant une des deux libertés de la colonne e , elle implique encore le placement de la cinquième en $(VIII, e)$. Mais la case (II, b) était elle-même obligatoire et (VII, h) l'est devenue. Ces dernières affectations forcent à leur tour la pose (I, g) qui est la seule encore permise et est compatible avec toutes les autres. Comme c'est la huitième dame, nous avons obtenu une des (douze) solutions fondamentales du problème (figure 5.18).

	a	b	c	d	e	f	g	h
I	3	6	1	2	3	1	[8]	2
II	1	[6]	1	3	1	2	2	4
III	2	1	1	1	2	[2]	2	2
IV	1	1	[1]	1	1	1	1	1
V	[3]	1	1	1	3	2	3	2
VI	1	3	1	[4]	1	2	4	4
VII	3	2	1	4	4	1	6	[7]
VIII	2	4	1	3	[5]	2	1	5

FIG. 5.18 – Une solution complète : $[i]$: choix i , $[j]$: implication j .

On peut noter cette solution de façon abrégée en ne retenant que les indices des colonnes dans l'ordre naturel des lignes : (g, b, f, c, a, d, h, e) . Cette solution ne dépend que des quatre premiers choix : si nous voulons toutes les solutions au problème des huit dames, il nous faut revenir sur chacun d'eux l'un après l'autre.// Revenant d'abord sur le troisième choix, nous interdisons la case $(V, 1)$ en la marquant d'un 2, puisqu'elle est maintenant liée à la seconde hypothèse. La case suivante est la case (V, e) : en

marquant les interdictions, nous constatons que la case (II, d) est à ce stade forcée, et que, dans ces conditions, aucune case de la ligne $VIII$ ne peut convenir : c'est ce que montre la figure 5.19.

	a	b	c	d	e	f	g	h
I	3		1	2	4	1		2
II	1	3	1	4	1	2	2	3
III	2	1	1	1	2	2	2	2
IV	1	1	1	1	1	1	1	1
V	2	1	1	1	3	2	3	2
VI	1		1	3	1	2		
VII		2	1	4	3	1	3	
VIII	2	3	1	4	3	2	1	3

FIG. 5.19 – Impossibilité après trois choix

Revenons encore une fois sur nos pas : la dernière possibilité après les choix 1 et 2 est de placer 3 en (V, g) . Il y a alors plusieurs lignes avec un degré de liberté égal à deux ; choisissant la II nous plaçons 4 en (II, b) : 5 en (VII, a) est alors forcé, mais ces deux derniers placements interdisent toutes les possibilités en dernière colonne. Enfin l'autre case disponible en ligne II soit (II, h) implique dans l'ordre : $(VI, 2)$, (I, a) et (VII, d) , mais toutes les cases sont alors interdites et il n'y a plus de solution (figure 5.20).

	a	b	c	d	e	f	g	h
I	6	5	1	2		1	3	2
II	1	4	1	3	1	2	2	4
III	2	1	1	1	2	2	2	2
IV	1	1	1	1	1	1	1	1
V	2	1	1	1	2	2	3	2
VI	1	5	1	4	1	2	3	3
VII	5	2	1	7	3	1	3	4
VIII	2	4	1	3	7	2	1	4

FIG. 5.20 – Fin de la recherche après deux choix

Nous avons donc démontré l'existence et l'unicité de la solution à partir de la configuration obtenue après les deux premiers choix (IV, c) et (III, f) correspondant à la solution partielle : $y = (., ., f, c, ., ., ., .)$. Cette configuration a été prise absolument au hasard. Nous pouvons en déduire une évaluation de l'effort de recherche total pour construire toutes les solutions. La première dame, une fois posée, laisse cinq choix possibles pour la seconde, quand celle-ci est placée sur une des lignes adjacentes, sauf si la première dame est placée sur une des cases extrêmes. Dans ce cas, la seconde dispose de six choix. Or, en tenant compte de la symétrie de l'échiquier, et donc de la symétrie des solutions, par rapport à l'axe vertical médian, on peut réduire l'étude aux quatre seuls cas où la première dame est à gauche de cet axe. Si bien qu'en définitive, nous aurons à répéter une recherche grossièrement équivalente à celle que nous avons menée pour la configuration (IV, c) , (III, f) , un nombre de fois égal à $6 + 5 + 5 + 5 = 21$. Ce nombre est remarquablement faible. On voit qu'avec de bons critères pour effectuer les choix et une bonne gestion des implications et retours arrière, trois choix sont pratiquement suffisants pour conclure.

Le troisième choix ne porte en moyenne que sur trois possibilités : le nombre total de situations est donc de l'ordre de $3 \times 21 = 63$ (presque chacune d'entre elles fournit une solution), alors qu'au début une permutation quelconque pouvait être envisagée. Nous sommes passés de $8! = 40320$ cas à 63 cas, soit un gain d'un facteur 640. Si nous ne voulons conserver que les solutions fondamentales, il conviendra finalement de trier l'ensemble obtenu et d'éliminer toute solution qui se déduit d'une autre par symétrie horizontale ou rotation. Il restera alors finalement douze solutions parmi lesquelles seule la première est complètement symétrique. L'algorithme auquel nous parvenons finalement effectue une énumération implicite de l'espace de recherche Y factorisé en ses composants $Y_1, Y_2, \dots, Y_k, \dots, Y_n$ en effectuant les choix sur les Y_i dans un ordre non fixé à l'avance. De plus, cet algorithme possède une caractéristique remarquable : il est tout à fait général et convient pour un problème combinatoire fini absolument quelconque. Nulle part, en effet, n'intervient l'énoncé spécifique du problème des huit dames.

Ce schéma unique reste donc valable quand on passe à des énoncés d'apparences assez différentes : nous allons tour à tour l'appliquer au problème de la coloration optimale d'un graphe, au problème du voyageur de commerce, à une mécanisation raisonnable du raisonnement, au jeu d'échecs.

5.9 Vers la programmation dynamique

La *programmation dynamique* est une méthode de résolution de problèmes d'optimisation. Le mot programmation ne doit pas être pris dans le sens qu'on lui donne en informatique : c'est un dérivé des termes « *programme mathématique* » qui désignent des systèmes de contraintes que l'on veut résoudre. L'idée essentielle de la programmation dynamique est de considérer les inconnues du programme mathématique comme des variables de décision pour lesquelles des valeurs vont être **choisies séquentiellement**. Si deux jeux différents de décisions conduisent à la même situation, seul le meilleur d'entre eux est conservé. La recherche est menée en étudiant en parallèle toutes les valeurs pertinentes pour chacune des variables. Le processus est donc de type **énumératif implicite en largeur d'abord**.

Exemple 1 :

« Un artisan doit honorer, dans les trois mois qui viennent, une commande de quatre exemplaires d'un même meuble. Tenant compte des coûts de réglage des appareils, des salaires des compagnons, des coûts de stockage, il calcule le coût global de fabrication de x_j meubles ($0 \leq x_j \leq 4$) au cours du mois j ($1 \leq j \leq 3$) pour tous les couples (j, x_j) envisageables. Il dresse ainsi le tableau suivant :

		Nombre d'exemplaires :				
		0	1	2	3	4
Mois de fabrication :	1	10	12	15	20	-
	2	10	14	16	19	23
	3	10	10	17	19	-

Il cherche à minimiser la somme des coûts $c(i, j)$ sur l'ensemble des trois mois en respectant la commande c'est-à-dire la contrainte : $x_1 + x_2 + x_3 = 4$. »

Par rapport à l'énumération pure, l'économie dans l'effort de recherche provient d'une étude pas à pas, mois après mois : à la fin du premier mois le coût des quatre situations possibles (0, 1, 2 ou 3 meubles fabriqués) est directement lu sur la première ligne du

tableau. A la fin du second mois, le cumul des coûts des deux premières lignes donne de la même façon le coût des cinq situations réalisables. Mais il est clair qu'indépendamment de ce qui sera ensuite décidé, seul est à considérer le plus petit coût pour chacune d'entre elles. Donc :

$$\text{coût } (s \text{ exemplaires fabriqués à la fin du mois 2}) = \min \text{ coût } (y \text{ à la fin du mois 1}) + \text{coût } ((s - y) \text{ au cours du mois 2})$$

Il vient ainsi à la fin du deuxième mois :

$$\text{coût } (s = 0) = 10 + 10 = 20 \quad (1)$$

$$\text{coût } (s = 1) = \min(12 + 10, 10 + 14) = 22 \quad (2)$$

$$\text{coût } (s = 2) = \min(15 + 10, 12 + 14, 10 + 16) = 25 \quad (3)$$

$$\text{coût } (s = 3) = \min(20 + 10, 15 + 14, 12 + 16, 10 + 19) = 28 \quad (4)$$

$$\text{coût } (s = 4) = \min(20 + 14, 15 + 16, 12 + 19, 10 + 23) = 31 \quad (5)$$

« Pour que la décision globale soit optimale, il faut que tout sous-ensemble de décisions soit déjà optimal ». Nous préciserons un peu plus loin ce théorème que nous venons d'utiliser. Puisque la quantité totale fabriquée au bout des trois mois est imposée, la solution est obtenue en complétant à quatre unités, chacune des situations atteintes :

$$\text{coût } (s = 4 \text{ à la fin}) = \min(31 + 10, 28 + 12, 25 + 17, 22 + 19) = 40 \quad (6)$$

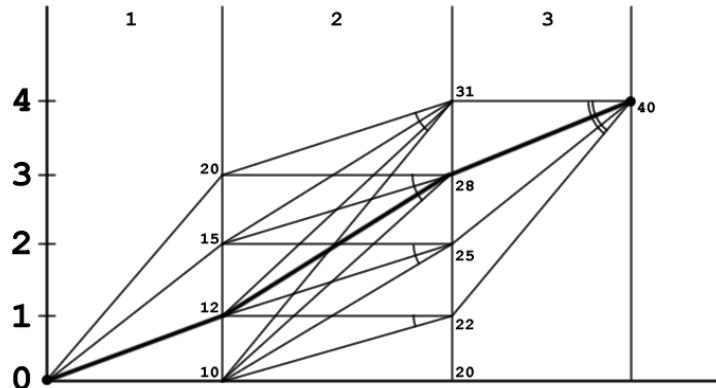


FIG. 5.21 – Graphe des décisions en programmation dynamique

Le coût global optimal de fabrication des quatre meubles est donc 40. Les variables de décision auxquelles il correspond peuvent être retrouvées en remontant les calculs : $x_3 = 1$ d'après (6), donc à la fin du mois 2, on avait $s = 3$; Il vient alors d'après (4) : $x_2 = 2$ et enfin : $x_1 = 1$. Le « graphe des décisions » de la figure 5.21 donne une image schématique de la résolution du problème.

5.9.1 Théorie de la programmation dynamique

Le processus d'optimisation séquentielle qui vient d'être utilisé est applicable à toute une famille de problèmes. Il suffit que des conditions très générales sur la fonction à optimiser soient respectées.

Définition : Soit une fonction f de \mathbb{R} dans \mathbb{R} des variables u, v et w : f est *décomposable* si :

a) il existe deux fonctions g et h , telles que l'on puisse écrire :

$$\forall(u, v, w), f(u, v, w) = g(u, h(v, w))$$

b) la fonction g ainsi définie est monotone non décroissante par rapport à sa seconde variable :

$$\forall(u, y_1, y_2), y_1 > y_2 \Rightarrow g(u, y_1) \geq g(u, y_2)$$

Toute la programmation dynamique repose alors sur le résultat suivant.

Théorème d'optimalité

Pour toute fonction f décomposable de \mathbb{R}^3 dans \mathbb{R} avec : $f(u, v, w) = g(u, h(v, w))$, on a :

$$OPT[f(u, v, w)] = OPT[g(u, OPT[g(v, w)])]$$

Ce théorème ramène le calcul d'un optimum d'une fonction de trois variables aux calculs de deux optimums successifs de fonctions de deux variables. On trouvera une démonstration par exemple dans Laurière 1979. Ce théorème est le plus souvent utilisé pour des fonctions d'optimisation $z(x_1, x_2, \dots, x_n)$ de n variables qui sont de la forme :

$$z(x_1, x_2, \dots, x_n) = R_1[r_1(x_1), z_1(x_2, x_3, \dots, x_n)]$$

avec R_1 décomposable et z_1 du même type que z , c'est-à-dire elle-même décomposable. On peut alors écrire :

$$OPT z = OPT[R_1[r_1(x_1), OPT R_2[r_2(x_2), \dots, OPT[r_n(x_n)] \dots]]]$$

ceci donne directement le schéma de la programmation dynamique, qui est donc caractérisée comme une méthode de résolution de problèmes d'optimisation, bâtie sur des **équations de récurrence**. En particulier, lorsque z est une somme de coûts élémentaires d'une seule variable, comme dans l'exemple précédent, les hypothèses du théorème sont bien vérifiées.

Cette méthode est appliquée couramment avec succès à la gestion des stocks, au pilotage d'engins, à la conduite de processus industriels (particulièrement en chimie), à la planification économique, à la théorie des jeux et des phénomènes stochastiques.

5.9.2 Intérêt de la programmation dynamique

Un énoncé imagé du théorème fondamental dit que : « *Toute sous-stratégie d'une stratégie optimale ne peut être elle-même qu'optimale* », et est concrètement représenté par la figure 5.22.

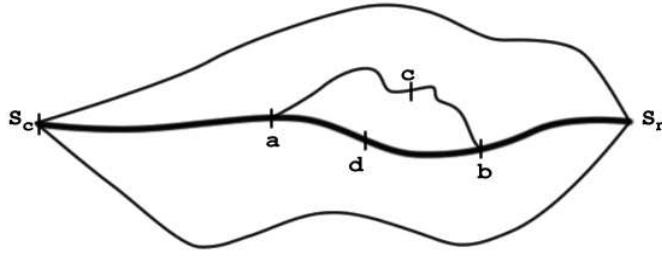


FIG. 5.22 – Stratégie optimale : la sous-stratégie (a, c, b) ne saurait être meilleure que (a, d, b) , sinon (S_0, a, d, b, S_n) ne pourrait être optimale.

Le gain est ainsi obtenu parce qu'à chaque étape intermédiaire, toutes les stratégies qui aboutissent à une situation donnée sont comparées ; seule la meilleure d'entre elles - ou plutôt sa valeur - est conservée pour la suite. Ainsi, supposons que chaque variable de décision x_i puisse prendre X_i valeurs : là où l'énumération exhaustive oblige à considérer :

$$\prod_{i=1}^n X_i \text{ situations,}$$

la programmation dynamique permet de ramener l'étude à :

$$\sum_{i=1}^n X_i \times S_i$$

cas différents, où S_i désigne le nombre de situations possibles avant le choix de la ième variable.

Le plus souvent les S_i sont bornés par la nature du problème ; ainsi, dans le premier exemple : $\forall i, S_i = 5$, et la programmation dynamique donne une complexité de calcul de l'ordre de $\sum X_i$. Dans certains cas, cette complexité est même un polynôme en fonction des données (plus court chemin, sac à dos, certains problèmes d'ordonnancements). Mais, un grand avantage de la méthode est de pouvoir également s'appliquer à des systèmes évoluant en avenir aléatoire : le futur n'est défini qu'en termes de probabilités et il faut cependant trouver la stratégie optimale.

Exemple 2 :

Gestion de stocks avec demandes incertaines. On doit commander et stocker un produit sur trois périodes de temps, de façon à maximiser l'espérance mathématique du bénéfice. Au départ le stock est au niveau zéro. Au début de chaque période i , on peut décider de l'achat de x_i unités au prix a_i . Durant la période i , une demande de y_i unités, au prix de vente v_i , s'effectue avec une probabilité $p_i(v_i)$ et, si le niveau du stock est trop faible, la partie non-satisfait de la demande ne cause pas de préjudice. A la fin de la troisième période les invendus éventuels sont soldés au prix de 20 l'unité. Les valeurs des vecteurs a , v , $p(v)$ sont estimées par la table suivante ; on a supposé $0 \leq x_i \leq 2$ et $1 \leq y_i \leq 2$ pour tout i .

période :	i	1	2	3
prix d'achat :	a_i	10	10	30
prix vente :	v_i	30	30	50
probabilités :	$p_i(1)$	0, 6	0, 6	0, 5
	$p_i(2)$	0, 4	0, 4	0, 5

Le système évolue entre 5 états possibles, puisque le nombre d'unités s_i en stock varie de 0 à 4. Entre chaque phase de décision, vient maintenant s'intercaler une phase aléatoire qui fait passer de l'état s à l'état $\max(0, s_i - y_i)$ avec une probabilité $p(y_i)$. Si le bénéfice

dans la situation s_i , après la décision x_i , est $c(s_i)$, son espérance mathématique $\bar{c}(w_i)$ en fonction de la demande est donc :

$$\bar{c}(w_i) = c(s_i) + \sum_{y_i=1}^z p(y_i) \times v_i \times w_i$$

avec $w_i = \max(0, s_i - y_i)$.

C'est cette quantité qui doit être récursivement optimisée. Les calculs seront ici plus simples en partant de la fin (période 3), puisqu'alors tout est calculable. Il vient :

$$\bar{c}_3(0) = \max [0, -30 + 50, -60 + (100 \times 0,5 + 70 \times 0,5)] = 25$$

$$\bar{c}_3(1) = \max [50, -30 + (100 \times 0,5 + 70 \times 0,5), -60 + (120 \times 0,5 + 90 \times 0,5)] = 55$$

$$\bar{c}_3(2) = \max [85, -30 + (120 \times 0,5 + 90 \times 0,5), -60 + (140 \times 0,5 + 110 \times 0,5)] = 85$$

puis, en remontant par récurrence :

$$\bar{c}_2(0) = \max [25, -10 + 30 + 25, -20 + (30 + 55) \times 0,6 + (60 + 25) \times 0,4] = 65$$

$$\begin{aligned} \bar{c}_2(1) = \max [30 + 25, -10 + (60 + 25) \times 0,4 + (30 + 55) \times 0,6, & \quad \text{car le} \\ & \quad -20 + ((30 + 85) \times 0,6 + (60 + 55) \times 0,4)] = 95 \end{aligned}$$

stock maximal, à la fin de la première période, vaut 2 et la demande au moins 1. Enfin, pour la première période, avec un stock initial nul, on a :

$$\bar{c}_1(0) = \max [65, -10 + 30 + 25, -20 + ((60 + 65) \times 0,4 + (30 + 95) \times 0,6)] = 105$$

La stratégie optimale consiste à acheter 2 unités à la première période, 2 unités à la seconde, et enfin 2, 1 ou 0 unité(s) à la troisième suivant que le stock est alors de 0, 1 ou 2 unités. En suivant cette stratégie, l'aléa de la vente fait évoluer le stock de plusieurs façons différentes, mais, dans tous les cas, le résultat moyen optimal de 105 est assuré.

5.10 Coloration optimale des sommets d'un graphe

Coloriage des cartes de géographie

Le problème de la coloration d'une carte de géographie est un problème très célèbre : il s'agit, étant donnée une carte tracée sur un plan ou sur une sphère et représentant différents pays avec leurs frontières, de parvenir à colorier chaque pays avec une couleur à déterminer, de telle sorte que deux pays limitrophes ne reçoivent en aucun cas la même coloration.

Ce problème est justement célèbre car il a défié les mathématiciens une centaine d'années ; on cherchait à démontrer le résultat suivant, longtemps connu sous le nom de *conjecture des quatre couleurs* (une conjecture est un théorème qu'on suppose vrai, qui est vérifiée pratiquement sur un grand nombre de cas mais dont la démonstration formelle complète n'a pu être établie) : « **Il est possible de colorier toute carte sur le plan ou sur la sphère en quatre couleurs seulement** ».

Finalement le théorème a été prouvé en 1976 avec l'aide d'un ordinateur qui a examiné de façon purement combinatoire un nombre énorme de cas de figures. [Appel et Haken 1976]. La première étape, pour étudier ce problème, est de simplifier la représentation et de se ramener à l'étude d'un dessin simple, appelé graphe non orienté (Cf. les problèmes du chapitre précédent). On obtient celui-ci en particularisant, dans chaque pays, un point intérieur, la capitale par exemple, et en reliant deux points par un trait appelé arête à chaque fois que leurs pays d'origine sont voisins. Il s'agit alors de colorier les points, appelés sommets, dans ce dessin, en respectant la règle du jeu :



FIG. 5.23 – Carte de l'Europe

deux sommets reliés par une arête doivent être coloriés différemment.

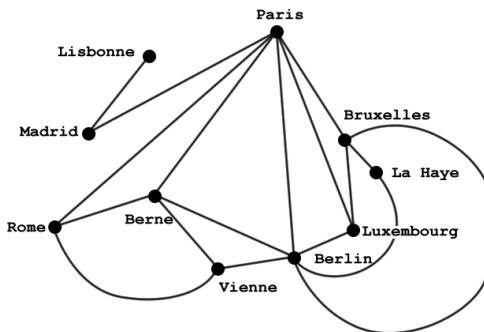


FIG. 5.24 – Graphe équivalent à la carte 5.23

Sous cette forme, on voit que le problème peut être généralisé : « Etant donné cette fois un graphe **quelconque** c'est-à-dire un ensemble de sommets et un ensemble d'arêtes reliant ces sommets deux à deux, trouver le plus petit nombre de couleurs associées aux sommets en respectant la même règle, soit : deux sommets reliés par une arête sont coloriés de deux couleurs différentes. » Le graphe n'est plus ici nécessairement « planaire » comme il l'était par construction dans le cas précédent. Ceci signifie qu'il n'existe pas nécessairement de représentation plane d'un graphe telle que deux arêtes quelconques ne se rencontrent pas.

C'est sous cette forme que l'on rencontre le problème dans différents contextes comme par exemple dans l'exemple suivant.

5.10.1 Organisation d'un colloque

Le secrétariat chargé d'établir le calendrier des sessions d'un colloque, a pris le soin de demander à chaque participant la liste des sessions auxquelles il voulait assister : le sous-ensemble de sessions ainsi associé à chaque participant devra être distribué sur des périodes toutes disjointes les unes des autres au cours du temps.

Exemple :

On a neuf séances, nommées a, b, c, \dots, i , qui durent toutes une demi-journée. On dispose d'un nombre de salles aussi grand que nécessaire. Les sous-ensembles de sessions ($ae, bc, cd, ed, abd, ahl, bhl, dfl, dhl, fgh$) donnent les incompatibilités : il existe au moins une personne qui désire participer à toutes les sessions de chaque sous-ensemble. Le problème est immédiatement ramené à un problème de coloration en dessinant le graphe dont les sommets correspondent aux sessions et les arêtes aux incompatibilités entre sessions comme le montre la figure 5.25. Et il s'agit maintenant pour le secrétariat de faire tenir au total le colloque dans le nombre minimal de demi-journées.

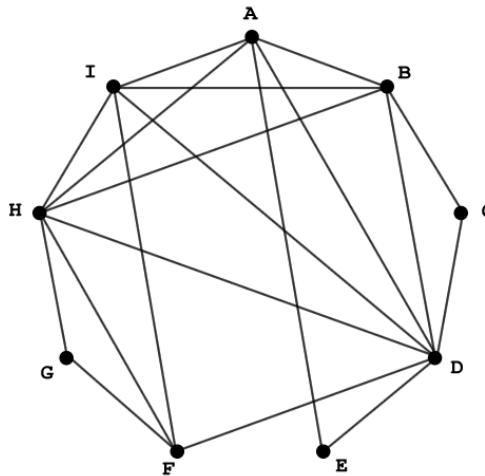


FIG. 5.25 – Graphe pour le problème du colloque

5.11 Algorithme de coloration optimale d'un graphe

Le nombre minimal de couleurs nécessaires pour colorier les sommets d'un graphe en donnant deux couleurs différentes à deux sommets voisins est appelé *nombre chromatique* du graphe G et noté $\gamma(G)$. La recherche de ce *nombre chromatique* est un problème combinatoire puisque en notant :

S : l'ensemble des sommets du graphe ;

U : l'ensemble des arêtes du graphe c'est à dire

des couples (i, j) du produit cartésien $S \times S$ tels que i est lié à j ;

$[1, q]$: l'ensemble des q premiers entiers naturels c'est à dire des q couleurs utilisées.

Ce problème s'écrit encore :

Trouver une fonction « couleur » : $S \rightarrow [1, q]$, telle que pour tout (i, j) de U , on ait $\text{couleur}(i) \neq \text{couleur}(j)$, telle que q soit minimum.

Nous pouvons procéder comme pour le problème des huit dames : coloriant d'abord un sommet, en tenant compte ensuite des implications auxquelles donne naissance cette affectation, on continue ainsi par énumération implicite tant que tous les sommets n'ont pas reçu de coloration. La différence essentielle tient en fait à l'optimisation : il ne s'agit plus ici de s'arrêter quand on a épousé le champ du possible, mais bien avant, dès que l'on peut prouver qu'il ne saurait exister de solution meilleure que la meilleure solution connue. Le critère d'arrêt est ainsi lié à la démonstration de l'impossibilité d'existence

d'une solution pour un certain sous-problème. La démarche générale devient :

- Trouver une solution en q couleurs ;
- Améliorer cette solution :
 - | TANT QUE on ne peut pas prouver que le nombre chromatique cherché est au moins égal à q , REPETER :
 - | $q \leftarrow q - 1$
 - | Chercher une nouvelle solution (en q couleurs)

Algorithme 5.A

5.11.1 Première étape : recherche d'une solution

Trouver une première coloration est un problème facile puisqu'il suffit de prendre pour la fonction couleur l'identité : $\forall i, i \in S : \text{couleur}(i) = i$. On obtient ainsi une coloration admissible puisqu'absolument toutes les couleurs des sommets sont différentes. Le nombre \bar{q} de couleurs utilisées est alors égal à la cardinalité de S .

Lorsque nous devons ensuite améliorer cette solution, nous sommes devant un problème d'existence exactement semblable à celui des huit dames : chaque sommet (au lieu de chaque ligne de l'échiquier) possède au départ q libertés (au lieu de huit) avec $\bar{q} > q$. L'algorithme ENUM est donc à nouveau utilisable. Il faut simplement préciser :

- a) comment se fait le choix en cas d'égalité des degrés de liberté ;
- b) comment se fait la gestion des implications.

Le point a) est réglé en reprenant une idée qui a déjà fait ses preuves et avait donné justement naissance aux tableaux de liberté : il y a toujours intérêt à effectuer le choix qui donne le plus d'information. Ceci revient à considérer ici le sommet le plus contraint, c'est-à-dire celui qui intervient dans le plus de contraintes. Mais on rencontre ici un seul type de contrainte qui est :

Pour tout (i, j) de U : $\text{couleur}(i) \neq \text{couleur}(j)$

Le sommet de S le plus contraint est donc celui dont sont issues le plus grand nombre d'arêtes de U . On appelle degré d'un sommet s dans un graphe $G = (S, U)$ le nombre d'arêtes passant par s : on choisira donc, à nombres de liberté égaux, le sommet non encore colorié qui est de degré maximum dans le graphe restreint aux sommets non encore coloriés. Comme d'habitude, ce critère n'est pas absolu (il pourrait être changé), ni indispensable (en son absence l'algorithme reste valide), il n'est là que par souci d'efficacité. Le point b) est ensuite particulièrement facile à traiter puisque la donnée des arêtes du graphe permet de garder en mémoire de façon explicite les interdictions. Bien sûr, comme pour les huit dames, il conviendra de prendre en compte les implications en chaîne, les affectations forcées (une seule couleur disponible) et les cas d'impossibilité, avec les retours arrière et le rétablissement des libertés qu'ils entraînent.

A la vérité, arrivé à ce point, il vous apparaît sans aucun doute clairement qu'il nous est possible de construire, sans aucun risque d'échec, une solution meilleure que la solution triviale obtenue avec : couleur = identité. Il suffit pour cela de ne considérer que le critère sur les degrés dans le graphe que nous venons de mettre en place en a). Les nombres de libertés sont en effet a priori tous égaux et le critère les concernant n'est plus pertinent. On construit alors une « bonne » solution en exécutant l'algorithme « *glouton* » suivant.

TANT QUE tous les sommets n'ont pas reçu de couleur : REPETER
colorier le sommet relié au plus grand nombre de sommets non coloriés,
en déduire les interdictions

Algorithme 5.A

Exemple 1 :

Pour le graphe de la carte géographique, nous avons au départ les degrés :

sommets	L	M	P	Br	LH	Bn	Lx	V	Be	R
degrés	1	2	6	4	2	6	3	3	4	3

et on colorie d'abord le sommet P , qui est un de ceux de plus grand degré, de la première couleur disponible :

$$I \rightarrow \text{couleur}(P)$$

On interdit ensuite la couleur I aux six sommets adjacents en diminuant en même temps leur degré d'une unité. On obtient ainsi la deuxième ligne du tableau 5.2. Le sommet de degré le plus élevé est maintenant Bn ; Il reçoit la couleur II , puisque I n'est pas disponible ici :

$$II \rightarrow \text{couleur}(Bn)$$

Les degrés sont modifiés comme indiqué en troisième ligne du tableau 5.2. Choisissant alors le premier sommet de degré 2, il vient :

$$III \rightarrow \text{couleur}(Br)$$

car ce pays compte déjà un voisin I et un voisin II . On obtient les degrés de la quatrième ligne du même tableau. On peut faire alors également :

$$III \rightarrow \text{couleur}(V)$$

puisque cette troisième couleur n'est pas interdite pour V . La fin est immédiate :

$$\begin{aligned} I &\rightarrow \text{couleur}(L) \\ II &\rightarrow \text{couleur}(M) \\ I &\rightarrow \text{couleur}(LH) \\ III &\rightarrow \text{couleur}(Lx) \\ IV &\rightarrow \text{couleur}(Be) \\ II &\rightarrow \text{couleur}(R) \end{aligned}$$

Nous avons obtenu une solution en quatre couleurs. Mais actuellement, rien ne prouve que l'on ne peut pas faire mieux, nous savons seulement que pour cette carte :

$$\gamma \leq 4$$

Ce que donnait d'ailleurs directement le théorème de Appel et Haken.

	L	M	P	Br	LH	Bn	Lx	V	Be	R
degrés résiduels	1	2	6	4	2	6	3	3	4	3
	1	1	-	3	2	5	2	3	3	2
	1	1	-	2	1	-	1	2	2	1
	1	1	-	-	0	-	0	2	2	2
	1	1	-	-	0	-	0	-	1	1
couleurs interdites	I		I		I	I		I	I	
		II	II		II		II		II	
				III	III					

TAB. 5.2 – Coloration d'une carte de géographie

Exemple 2 :

Pour le graphe du colloque, nous avons au départ les degrés donnés dans la première ligne du tableau 5.3. On fait donc en premier lieu :

$$I \rightarrow \text{couleur}(d)$$

Ensuite il vient successivement, en se servant des résultats intermédiaires consignés dans le tableau 5.3 :

$$\begin{aligned} II &\rightarrow \text{couleur}(h) \\ III &\rightarrow \text{couleur}(a) \\ IV &\rightarrow \text{couleur}(b) \\ III &\rightarrow \text{couleur}(f) \end{aligned}$$

	a	b	c	d	e	f	g	h	i
degrés résiduels	5	5	2	7	2	4	2	6	5
	4	4	1	-	1	3	2	5	4
	3	3	1	-	1	2	1	-	3
	-	2	1	-	0	2	1	-	2
	-	-	0	-	0	2	1	-	1
	-	-	0	-	0	-	0	-	0
couleurs interdites	I	I	I	-	I	I	-	I	I
	II	II			II	II		II	
	III			III			III		
		IV					IV		
							III		

TAB. 5.3 – Coloration pour le problème du colloque

Puis, finalement, alors que les sommets restants sont tous de degrés nuls :

$$\begin{aligned} II &\rightarrow \text{couleur}(c) \\ II &\rightarrow \text{couleur}(e) \\ I &\rightarrow \text{couleur}(g) \\ V &\rightarrow \text{couleur}(i) \end{aligned}$$

Nous pouvons alors affirmer que pour ce deuxième graphe on a :

$$\gamma' \leq 5$$

Nous avons ainsi effectué la première étape de l'algorithme initial, proposé en 5.A. Il faut maintenant, « tant qu'on n'a pas pu prouver que le nombre chromatique cherché

était égal à q , faire $q \leftarrow q - 1$ et chercher une nouvelle solution avec ce nombre de couleurs ».

5.11.2 Deuxième étape : recherche de l'optimalité

Puisque nous savons que chaque sommet est maintenant limité à q libertés, nous faisons ici intervenir des vecteurs L_i de q composantes. Mais en réalité, nous pouvons aller un peu plus vite. Il convient en effet de tenir compte de ce que toute solution n'est réellement définie qu'à une permutation des couleurs près : une couleur comme I , II ou III n'est, pour ce problème, qu'un nom qui sert de repère. Ainsi, lorsque nous avons commencé à dérouler l'algorithme 5.B, les premières affectations telles :

dans l'exemple 1 :	dans l'exemple 2 :
$I \rightarrow \text{couleur}(P)$	$I \rightarrow \text{couleur}(d)$
$II \rightarrow \text{couleur}(Bn)$	$II \rightarrow \text{couleur}(h)$
$III \rightarrow \text{couleur}(Br)$	$III \rightarrow \text{couleur}(a)$
	$IV \rightarrow \text{couleur}(b)$

n'étaient pas de vrais choix mais de simples **conventions** sur le nom des couleurs des premiers sommets qui, en tous cas, devraient être coloriés tous différemment. Or, si ces sommets doivent tous être coloriés différemment, c'est pour une raison simple : **ils sont tous reliés mutuellement deux à deux**. Dans l'exemple 1, P est lié à la fois à Bn et à Br qui lui-même est relié à Bn . Dans le second exemple d est lié à h , à a et à b ; h est lié à a et à b et enfin a est lui-même lié à b .

Ces sous-ensembles de sommets sont donc effectivement reliés de toutes les manières a priori possibles. Une nouvelle notion s'est donc dégagée très naturellement : on nomme de tels sous-graphes des *sous-graphes complets* ou encore des *cliques*. Par définition de la coloration des sommets d'un graphe on a donc la propriété :

Propriété : Il faut exactement p couleurs pour colorier une clique de p sommets. Ainsi, dans l'exemple 1, on sait maintenant que :

$$3 \leq \gamma \leq 4$$

puisque les sommets P , Bn , Br forment une clique de cardinalité 3. Mais en y regardant de plus près, le seul sommet qui a dû être colorié d'une quatrième couleur, à savoir Lx , l'a été parce qu'il était lui-même complètement relié à ces trois sommets. Le sous-graphe de sommets P , Bn , Br , Lx est bien, en effet, une nouvelle clique de cardinalité 4. Nous sommes dès lors en droit de conclure directement, pour cet exemple, que le nombre minimum de couleurs nécessaires à la coloration des sommets est 4.

$$\gamma = 4$$

Dans le second exemple un raisonnement en tous points semblable permet d'exhiber une clique de cinq sommets, a b d h i , montrant là aussi, pour ce graphe non planaire, que la première coloration était en fait optimale :

$$\gamma' = 5$$

Après ces deux succès, l'idée qui vient à l'esprit est tout simplement de se demander si l'on n'est pas tout bonnement devant un faux problème. N'a-t-on pas toujours l'égalité entre la cardinalité maximum d'une clique et la valeur minimale du nombre

de couleurs ? Un premier élément de réponse est de remarquer que de toutes façons le problème n'en serait pas pour autant résolu : qui va nous assurer qu'une clique donnée est bien la plus grande ? En fait, ce problème est lui-même aussi complexe que le problème de coloration initial.

Mais en outre, n'est-il pas possible de construire des graphes qui ne vérifient point cette propriété ? Or, très vite, un bricolage simple sur des assemblages de triangles (cliques d'ordre 3) conduit au contre-exemple de la figure 5.26 :

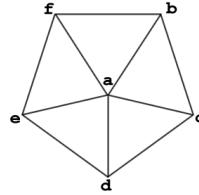


FIG. 5.26 – Un contre-exemple

Dans ce graphe, où la cardinalité maximum d'une clique, cardinalité que nous désignerons dorénavant par C_{max} pour abréger, est de toute évidence 3, on a en effet $\gamma = 4$, γ strictement supérieur à C_{max} . Ainsi, si l'on colorie a par I, b par II, c par III et si l'on veut une coloration en 3 couleurs seulement on obtient une suite d'implications dans l'algorithme 5.A : d est colorié par II, e par III mais f est ainsi relié à a de couleur I, à b de couleur II et enfin à e de couleur III ; l'hypothèse $\gamma = 3$ conduit donc à une absurdité.

Un deuxième contre-exemple, un peu moins immédiat, est donné par la figure 5.27 :

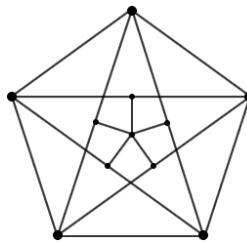


FIG. 5.27 – Deuxième contre-exemple

Cette fois on a un graphe dans lequel les seules cliques sont réduites à de simples arêtes, donc : $C_{max} = 2$, tandis qu'en déroulant l'algorithme 5.A complètement cette fois, ce qui oblige à de nombreux retours sur des choix antérieurs, on obtient :

$$\gamma = 4$$

On arrive donc à une différence de deux unités sur d'aussi faibles valeurs et sur des petits graphes. A la vérité, il existe des graphes que l'on peut construire de telle sorte que la différence $(\gamma - C_{max})$ soit aussi grande que tout entier fixé à l'avance. Finalement une seule chose est sûre :

Propriété 2 : $\gamma \geq C_{max}$.

La quantité C_{max} est une borne inférieure de l'optimum cherché. L'optimalité sera donc démontrée dans notre algorithme soit en démontrant par énumération implicite l'impossibilité de construire une coloration en q couleurs bien que q soit strictement

supérieur à C_{max} , soit, dans les cas favorables, parce qu'on aura pu exhiber effectivement une coloration en $q = C_{max}$ couleurs.

De toute manière il sera utile, et ce avant même la première tentative de coloration, de rechercher une clique de cardinalité C aussi grande que possible. Bien sûr nous ne pourrons être certains d'avoir obtenu C_{max} mais, comme en tous cas : $C_{max} \geq C$, nous aurons une borne inférieure valide. Pour faire cette recherche, une bonne idée est de remarquer que dans ce problème, en quelque sorte dual du problème de coloration, les degrés des sommets dans le graphe jouent encore un rôle : la clique est, dans un premier temps, à rechercher dans l'ensemble des sommets de plus grands degrés. Ayant obtenu une première clique de C sommets, on peut, tout comme dans le problème d'origine, chercher à l'améliorer : tout sommet de degré strictement inférieur à C peut être définitivement éliminé.

Pratiquement, l'algorithme final de coloration qui dérive directement de l'algorithme 5.C est devenu :

Recherche d'une "grande" clique Φ

$\Phi \leftarrow \text{vide} ;$

TANT QUE il existe des sommets non examinés
examiner le sommet s de plus grand degré dans G ;
s'il est relié par une arête à tout sommet de Φ

ALORS FAIRE : $\phi \leftarrow \Phi \cup \{S\}$

On a obtenu une clique de cardinalité C , ôter de G tous les
sommets de degrés strictement inférieurs à C

Recommencer dans le nouveau graphe la recherche d'une grande clique

TANT QUE le graphe obtenu est différent de celui de la
dernière itération et qu'il subsiste au moins $(C + 1)$ sommets
de degrés supérieurs ou égaux à C .

(Coloration de la clique) : $p \leftarrow 1$; $m \leftarrow 0$; (*nombre de sommets colorés*)

POUR TOUT sommet s de Φ FAIRE : $\text{Couleur}(s) \leftarrow p$; $p \leftarrow p + 1$;

(Initialisation)

$n \leftarrow (\text{nombre de sommets de } S)$; $k \leftarrow 0$ ($k = \text{nombre de choix effectués}$)

$L_i \leftarrow (0, 0, \dots, 0)$; $\text{Choix}(i) \leftarrow 0$; POUR TOUT $i \in S - \Phi$: $\text{Couleur}(i) \leftarrow 0$;

TANT QUE $k \geq 0$ REPETER

SI $m = n$ ALORS Imprimer

SINON $k \leftarrow k + 1$

Choisir le vecteur L_i qui contient et tel que $\text{Couleur}(i) = 0$

TANT QUE L_i contient au moins deux zéros REPETER :

$\text{Choix}(k) \leftarrow i$, $\text{Niveaux}(k) \leftarrow m$

Choisir une couleur y_i , tel que $L_i(y) = 0$

Enregistrer le choix : $\text{Couleur}(i) \leftarrow y$; $\text{Pile}(m) \leftarrow i$; $m \leftarrow m + 1$

Ôter y_i des choix possibles : $L_i(y_i) \leftarrow k$

Interdictions : Pour tout sommet s lié au sommet i et
non colorié : FAIRE $L_s(y_i) \leftarrow k$

(Implications)

Pour tout sommet i non colorié FAIRE :

SI L_i contient un seul zéro ALORS aller à Enregistrer

SI L_i ne contient aucun zéro ALORS :

(Retour)

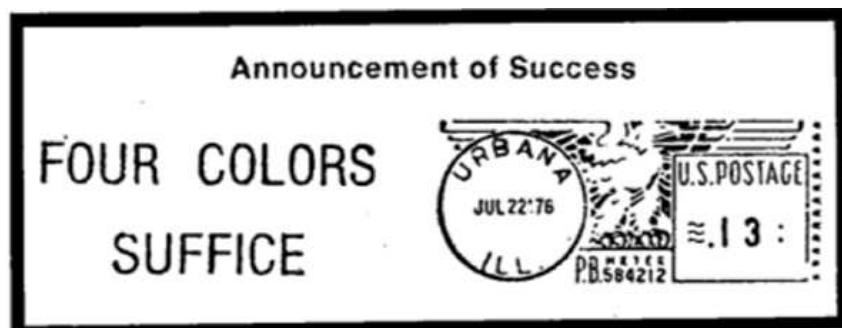
(Restaurer les libertés)

$\forall j, \forall i$ FAIRE SI $L_i(j) = k$ ALORS $L_i(j) \leftarrow 0$;
 $L_k(\text{Couleur}(\text{Niveaux}(k))) \leftarrow k - 1$

(Libérer les couleurs)

Pour $q = \text{Niveaux}(k)$ jusqu'à m FAIRE : $\text{Couleur}(\text{Pile}(q)) \leftarrow 0$

$m \leftarrow \text{Niveaux}(k)$; $k \leftarrow k - 1$



Avec la méthode qui vient d'être exposée vous êtes maintenant capable d'attaquer avec succès ce nouveau problème :

5.12 Le problème du voyageur de commerce

Ce problème dont l'énoncé est des plus simples, est aussi **un des plus célèbres problèmes de la Recherche Opérationnelle** : « *Un voyageur de commerce désire rendre visite à des clients situés dans n localités différentes ; il part de l'une d'entre elles et y revient à la fin de sa tournée. On suppose en outre que le voyageur ne passe jamais deux fois par une même localité. Dans quel ordre doit-il visiter ces localités, de façon à ce que la distance totale parcourue soit minimale ?* »

Il semble que les premières recherches sur ce problème aient été suscitées par un casse-tête dû au mathématicien anglais Hamilton en 1859 ; il s'agissait de dénombrer dans le graphe de la figure 5.28 tous les circuits différents que peut parcourir notre voyageur si les localités sont reliées comme indiqué.

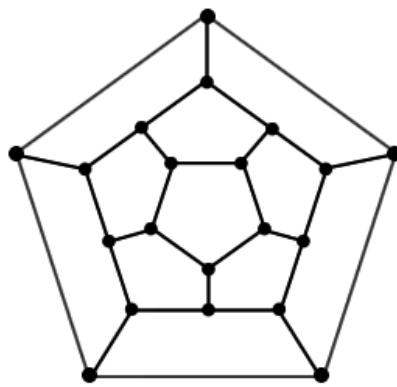


FIG. 5.28 – Le ”circuit autour du monde” d’Hamilton

Le jeu était une représentation symbolique du globe terrestre sous forme de dodécaèdre dont la figure donne le développement. Ce problème du « parcours fermé autour du monde » appelé également depuis problème du « *circuit hamiltonien* » possède des liens avec la théorie des groupes et également, de façon plus inattendue, avec le théorème des quatre couleurs donné au chapitre précédent.

Mais notre problème du voyageur de commerce fait, lui, intervenir en outre, les distances de ville à ville, qui sont supposées connues. Ces « distances » peuvent d'ailleurs aussi bien être changées en temps ou en coûts de parcours, ou même en des valuations absolument quelconques. Dans le cas général on ne supposera donc même pas que le coût pour aller de i en j est le même que pour le trajet inverse de j en i . Soit, par exemple, un voyageur qui doit visiter successivement 6 villes avec les coûts donnés par la figure 5.29.

	A	B	C	D	E	F
A	-	27	43	16	30	26
B	7	-	16	1	30	25
C	20	13	-	35	5	0
D	21	16	25	-	18	18
E	12	46	27	48	-	5
F	23	5	5	9	5	-

FIG. 5.29 – Matrice des distances pour un voyageur de commerce

La ville de départ étant fixée, il y a, a priori, autant de circuits différents que de permutations des $(n - 1)$ villes suivantes, soit $(n - 1)!$. Comme la coloration optimale d'un graphe, ce problème allie donc la simplicité d'énoncé et difficulté de résolution due à la dimension de l'univers de recherche. Comme pour la coloration, il est cependant trivial d'obtenir une solution au moins quand, comme ici, on peut se rendre de toute ville à toute autre ville. Comme pour la coloration enfin, nous aurons la démarche générale.

Trouver une solution de coût q ;

**TANT QU'il n'est pas prouvé que le coût q^* cherché est au moins égal à q ,
Chercher une meilleure solution (nouveau coût $q = q - 1$).**

5.12.1 Première étape : recherche d'une solution

Comme pour le problème précédent, il est trivial de trouver une solution, puisque n'importe quelle permutation des six villes convient. Ainsi le circuit $ABCDEF$ constitue une solution de coût 124. Remarquons que la méthode qui consiste ici à partir d'une ville et de progresser en empruntant toujours le coût le plus faible n'est certainement pas optimale et peut même conduire à de très mauvaises solutions. En effet, les contraintes $K(x)$, à savoir ici obtenir finalement un circuit unique, restreignent de plus en plus les choix à mesure que l'on progresse et, si les premiers coûts sont choisis faibles, les derniers coûts sont obligatoires et peuvent être arbitrairement grands.

Utilisons malgré tout ce procédé : partons de la ville C par l'arc CF de coût nul pour arriver en F : l'arc FC qui refermerait immédiatement le circuit nous est interdit. Nous quittons dès lors F par le premier arc autorisé, de coût 5, soit FB ; l'arc BC est interdit. On prend ici BD de coût 1 qui interdit OC . L'arc DE permet de partir de D , il interdit EC . EA nous conduit en A et l'arc AC ferme finalement le circuit de façon obligée. Son coût est déjà 43, mais il aurait tout aussi bien pu être encore plus grand. Nous retenons cependant ici la solution obtenue, $ACFBDEA$, de coût 79, qui améliore la précédente.

En construisant cette solution nous nous sommes contraints à former le circuit cherché de façon progressive en joignant à mesure les trajets bout à bout. Or, cette façon n'est nullement imposée : tout comme dans le problème des huit dames nous pouvons effectuer les choix dans un ordre absolument quelconque et fixer à chaque étape un arc arbitraire parmi l'ensemble de n arcs que nous devons en fin de compte retenir et ceci en respectant simplement la seule contrainte de $K(x)$, imposant l'existence d'un circuit unique : le sous-ensemble d'arcs retenu à un moment donné ne doit pas : arriver ou partir plus d'une fois en chaque ville : former de circuit, sauf à la fin, lorsque toutes les villes ont été traversées.

Dans cette optique, et forts de l'expérience précédente, nous voyons que plutôt que d'emprunter les arcs de coûts faibles, il convient de préférence de **ne pas emprunter les arcs de coûts trop élevés**. Quels sont donc les arcs que l'on peut raisonnablement qualifier de coûts élevés ? De toute évidence, cette notion est une notion relative : de façon plus précise, et parce qu'il faut en tout cas choisir un arc sur chaque ligne et un sur chaque colonne, il s'agira de tout arc dont le coût sera éloigné à la fois dans sa ligne et dans sa colonne des coûts minimaux. Décidons donc, mais c'est là une hypothèse de travail qu'il nous faudra peut-être remettre en cause, d'éviter de passer par tout arc qui, dans notre exemple, possède un coût supérieur à la fois au minimum en ligne augmenté de 10 et au minimum en colonne augmenté de la même quantité.

Dans le tableau suivant les coûts maxima autorisés sur chaque ligne et chaque colonne ont été indiqués et seuls, les arcs dont le coût est strictement inférieur à l'une de ces bornes, sont retenus.

	A	B	C	D	E	F		
A	-	-	-	16	-	-	26	Coûts maxima
B	7	-	-	1	-	-	11	en ligne
C	-	13	-	-	5	0	10	
D	21	16	25	-	18	18	26	
E	12	-	-	-	-	5	15	
F	-	5	5	9	5	-	15	
	17	15	15	11	15	10		Coûts maxima en colonnes

FIG. 5.30 – Elimination d'axes trop coûteux.

Si nous supposons donc que le circuit optimal cherché ne passe que par ces arcs encerclés, il est clair que nous faisons une hypothèse optimiste, puisque, à la vérité, rien ne nous permet d'affirmer que dans ce nouveau graphe, où seuls les « bons » parcours élémentaires subsistent, il existe encore seulement une solution ! (En effet, il aurait pu, par exemple, se faire que deux lignes données voient tous leurs arcs éliminés sauf un et que de plus les deux seuls arcs restants soient situés sur une même colonne. Alors, évidemment, on ne peut quitter ces deux villes que pour se rendre dans la ville unique correspondant à cette colonne et le problème du circuit hamiltonien n'a plus de solution).

Qu'en est-il ici ? Avons-nous été trop optimistes ? Le nouveau tableau nous indique qu'il n'y a plus qu'un degré de liberté pour la première ligne : la seule façon de quitter la ville *A* est de se rendre en *D*. Alors l'arc *DA*, qui conduit directement à un retour en *A*, doit être éliminé. De plus, comme nous avons déjà atteint *D*, les arcs *BD* et *FD* qui nous y mèneraient à nouveau sont maintenant à rejeter. Le nouveau tableau est devenu :

	A	B	C	D	E	F
A	-	-	-	16	-	-
B	7	-	-	-	-	-
C	-	13	-	-	5	0
D	-	16	25	-	18	18
E	12	-	-	-	-	5
F	-	5	5	-	5	-

L'arc *BA* est à son tour obligé et il a pour effet de supprimer les arcs *EA* et *DB*, ce dernier refermant en effet le circuit parasite *BAD*. La solution partielle peut être représentée par le petit schéma suivant. L'arc *EF* est alors obligé et il interdit le choix de *TE*.

A cette étape, tous les degrés de liberté en ligne et en colonne sont égaux à deux. Nous devons donc faire, pour l'arc suivant, un choix. Retenons, par exemple, l'arc de coût le moins cher pour la rangée qui coûte le plus cher, à savoir la ligne *D*. On choisit *DE* ; les arcs *DC* et *FE* sont interdits, tandis que les arcs *FC* et *CB* doivent obligatoirement être empruntés. Nous parvenons donc bel et bien à un circuit hamiltonien. C'est *ADEFBCA* et son coût, 64, est inférieur au meilleur coût trouvé précédemment. Nous avons ainsi mis au point une méthode rapide pour trouver un bon trajet admissible pour le voyageur de commerce.

Si l'élimination opérée sur l'ensemble des arcs en fonction de leurs coûts s'était révélée sévère au point de ne plus permettre de construire de circuit, il aurait simplement fallu

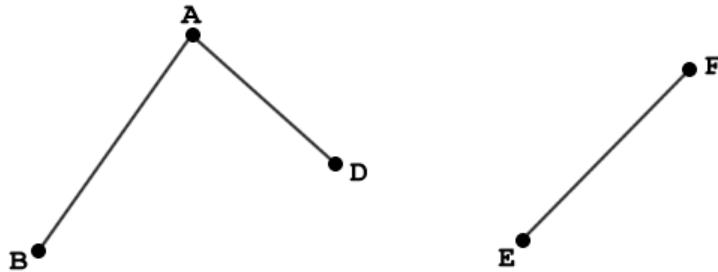


FIG. 5.31 – Solution partielle pour le voyageur

desserrer la contrainte ajoutée, c'est-à-dire conserver des arcs d'un coût relatif plus élevé par rapport aux minima. Ici, après plusieurs essais successifs, nous avons obtenu une « bonne solution ». Il convient maintenant de passer à la phase d'optimisation.

5.12.2 Seconde étape : recherche de l'optimum

Nous possédons une borne supérieure de l'optimum cherché. Comme lors d'une mesure physique, ou encore comme dans le problème de coloration, il est utile de compléter l'encadrement en déterminant une **borne inférieure** cette fois. Quel est donc le concept, analogue au concept de clique, qui nous permettra d'affirmer que le coût de toute solution vaut au moins une certaine quantité connue ? Raisonnons ici encore **en étudiant un sous-problème du problème d'origine** : à ce sous-problème plus petit et/ou plus simple correspondra une solution qui ne peut être que moins coûteuse que celle du problème initial. En particulier ici, considérons les villes une à une : il est nécessaire de toute façon, que le voyageur de commerce quitte chaque ville à un moment ou à un autre. Alors le coût de résolution de ce sous-problème - quitter la ville x - vaut au moins le coût minimal sur la ligne correspondante dans la matrice initiale. Ainsi, pour la ville A :

	A	B	C	D	E	F
A	-	27	43	16	30	26

Nous avons un coût minimal de 16 et, de plus, nous pouvons effectuer une translation de 16 unités sur toutes ces distances sans changer la forme du circuit optimal, puisque la fonction à optimiser est une simple somme des coûts élémentaires, à raison d'un coût par rangée. Les coûts de départ de la ville A deviennent alors :

	A	B	C	D	E	F
A	-	11	27	0	14	10

Or le même raisonnement est applicable indépendamment aux cinq autres lignes, de sorte qu'après les soustractions de 16 sur la ligne A , 1 sur B , rien sur C , 16 sur D , 5 sur E et 5 sur F , nous parvenons au nouveau tableau :

	A	B	C	D	E	F
A	-	11	27	0	14	10
B	6	-	15	0	29	24
C	20	13	-	35	5	0
D	5	0	9	-	2	2
E	7	41	22	43	-	0
F	18	0	0	4	0	-

Cependant, il faut également que le voyageur atteigne chacune des 6 villes exactement une fois ; ces passages correspondent cette fois aux 6 colonnes du tableau. Ils peuvent maintenant tous se faire en empruntant un arc de coût réduit nul, à l'exception du passage en colonne *A*, où l'arc le moins cher vaut encore 5 unités : cette quantité peut à son tour être soustraite, puisque nécessairement le sous-problème de l'arrivée en *A* doit être résolu. Les nouveaux coûts sont alors :

	A	B	C	D	E	F
A	-	11	27	0	14	10
B	1	-	15	0	29	24
C	15	13	-	35	5	0
D	0	0	9	-	2	2
E	2	41	22	43	-	0
F	23	0	0	4	0	-

Nous avons au total retranché une quantité q avec :

$$\underline{q} = 16 + 1 + 1 + 16 + 5 + 5 + 5, \text{ soit } \underline{q} = 48$$

Et nous savons que d'une part, toute solution avec les nouveaux coûts réduits doit être augmentée de cette quantité q pour redonner la valeur vraie q^* du circuit optimal cherché, cette valeur \underline{q} obéit ainsi à la relation d'encadrement :

$$\underline{q} \leq q^* \leq q_0$$

où q_0 désigne la meilleure valeur actuellement connue. De plus, puisque le circuit ne part et n'arrive qu'une fois en chaque ville, la longueur réelle du circuit à l'optimum vaut exactement q augmentée de la longueur du circuit optimal sur la matrice réduite. Dans notre exemple, nous avons actuellement : $48 \leq q^* \leq 64$. Ainsi, reprenant l'idée qui avait donné naissance aux premiers circuits, il va nous être possible d'éliminer les arcs dont les coûts sont trop élevés. Mais cette élimination va pouvoir être, cette fois, absolument rigoureuse. En effet considérons la différence $q_0 - \underline{q}$ qui vaut ici 16, elle correspond à l'écart « absolu » le plus grand entre la valeur cherchée et le circuit connu. Autrement dit, tout circuit d'un coût supérieur, au sens large, à cette différence est a priori à rejeter : il ne peut être optimal puisqu'il est déjà moins bon que le premier circuit connu. En particulier, tous les arcs qui possèdent un coût réduit supérieur ou égal à $q_0 - \underline{q}$ sont eux-mêmes à rejeter définitivement. La matrice des coûts du graphe des seuls trajets admissibles à ce stade devient donc :

	A	B	C	D	E	F
A	-	11	-	0	14	10
B	1	-	15	0	-	-
C	15	13	-	-	5	0
D	0	0	9	-	2	2
E	2	-	-	-	-	0
F	23	0	0	4	0	-

Il reste à construire sur cette matrice un nouveau circuit, si cela est possible, de coût inférieur à q_0 , puis à recommencer jusqu'à prouver l'optimalité.

Nous pourrions à nouveau utiliser les degrés de liberté et donc effectuer d'abord ici un choix relatif à la ligne E . Mais, parce que ce critère serait initialement inefficace pour une matrice de taille plus importante, nous allons utiliser une nouvelle idée, qui est liée aux coûts et qui mesure l'intérêt relatif des arcs de meilleurs coûts - les arcs de coûts nuls.

Ces arcs de coûts nuls ne sont pas, en effet, tous équivalents a priori. Un circuit idéal n'emprunterait que de tels arcs, mais, il n'y a aucune raison pour qu'un tel circuit existe : ici en particulier on ne peut par exemple à la fois partir de A et de B pour arriver en D . Ceci prouve d'ailleurs que $q^* \geq 49$, mais nous n'utiliserons pas directement ce résultat.

Certains arcs de coûts nuls ne pourront donc être retenus. La question est alors : **que coûte au minimum le fait de ne pas emprunter un arc de coût nul ?** Eh bien, puisqu'il faut, de toute manière, quitter la ville sur la colonne, le coût minimum de remplacement sera la somme du coût minimum en ligne, autre que le zéro concerné, et du coût minimum en colonne, autre que ce zéro. Cette quantité que l'on rencontre dans maints problèmes d'optimisation est habituellement appelée le *regret* associé au choix idéal consistant à prendre le coût nul. Elle mesure le coût le plus petit - regret minimum - qu'il faudra payer si l'on renonce à ce choix.

L'heuristique immédiate suivant cette idée sera bien sûr d'effectuer le choix dont le regret est maximum. Dans le problème du voyageur de commerce, le regret $r(i, j)$ d'un arc (i, j) de coût nul s'écrit :

$$r(i, j) = \min_{k, k \neq j} c(i, k) + \min_{l, l \neq i} c(l, j)$$

Ainsi, dans notre exemple : $r(D, A) = 0 + 1$, $r(D, B) = 0 + 0$, $r(F, B) = 0 + 0$, $r(F, C) = 0 + 9$, $r(A, D) = 10 + 0$, $r(B, D) = 0 + 1$, $r(F, E) = 0 + 2$, $r(C, F) = 5 + 0$, $r(E, F) = 2 + 0$.

Le choix qui coûterait le plus par la suite, si nous y renoncions maintenant, concerne donc l'arc AD . Prenant ainsi pour première hypothèse que le voyageur passe par l'arc AD , nous devons prendre en compte les implications liées à ce choix. Tous les arcs de la ligne A et de la colonne D ne doivent plus être considérés et l'arc DA lui-même qui impliquerait un circuit parasite doit être éliminé. Le tableau de travail est devenu celui de gauche :

	A	B	C	D	E	F		A	B	C	D	E	F
A							A						
B	1	-	15	-	-		B	0	-	14	-	-	-
C	15	13	-	5	0		C	15	13	-	5	0	
D	0	0	9	2	2		D	0	0	9	2	2	
E	2	-	-	-	0		E	2	-	-	-	0	
F	23	0	0	0	-		F	23	0	0	0	-	

Dans cette hypothèse, quitter la ville B coûte au moins une unité, si bien que nous effectuons à nouveau une translation des coûts sur cette ligne, et augmentons d'autant la borne inférieure $\underline{q}(AD)$ de tous les circuits empruntant l'arc AD : $\underline{q}(AD) = 49$.

Le tableau devient celui donné ci-dessus à droite. Le calcul des regrets donne : $r(B, A) = 16$, $r(D, B) = 2$, $r(F, B) = 0$, $r(F, C) = 9$, $r(F, E) = 2$, $r(C, F) = 9$, $r(E, F) = 2$. Il conduit à faire passer obligatoirement, après la première hypothèse, le voyageur par l'arc BA , puisque sinon le coût minimum du circuit serait $\underline{q}(AD) + r(B, A) = 49 + 16$, somme supérieure à 64, seuil limite autorisé pour améliorer la solution déjà connue.

Le passage par BA implique à son tour la suppression des autres arcs en colonne A

et ligne B , ainsi que celle de l'arc parasite DB . Il conduit à la nouvelle matrice donnée à gauche et réduite à droite, après soustraction de deux unités en ligne D , d'où $q(AD) = 51$.

	A	B	C	D	E	F		A	B	C	D	E	F
A							A						
B							B						
C	13		-	5	0		C	13	-		5	0	
D	0	9		2	2		D	0	7		0	0	
E	-	-		-	0		E	-	-		-	0	
F	0	0		0	-		F	0	0		0	-	

Le coût de l'arc CB , qui vaut 13, est maintenant, à lui tout seul, trop fort pour pouvoir faire partie d'un circuit d'un coût inférieur à celui que nous connaissons ; en effet :

$$\cot(CB) \geq q_0 - q(AD) = 64 - 51 = 13$$

Cet arc est donc éliminé. Cette fois il n'y a ensuite pas de choix : l'arc EF est seul sur sa ligne, il faut l'ajouter au circuit partiel actuel et supprimer l'arc FE . En outre, l'arc CE devient unique candidat (inutile donc de réduire la ligne d'abord), ce qui donne $q(AD) = 56$ et entraîne l'appartenance combinée de DC et de FB au circuit actuel. Finalement, nous obtenons pour un coût total de 63 unités le trajet $FBADCEF$. L'arbre de la recherche est celui de la figure 5.32.

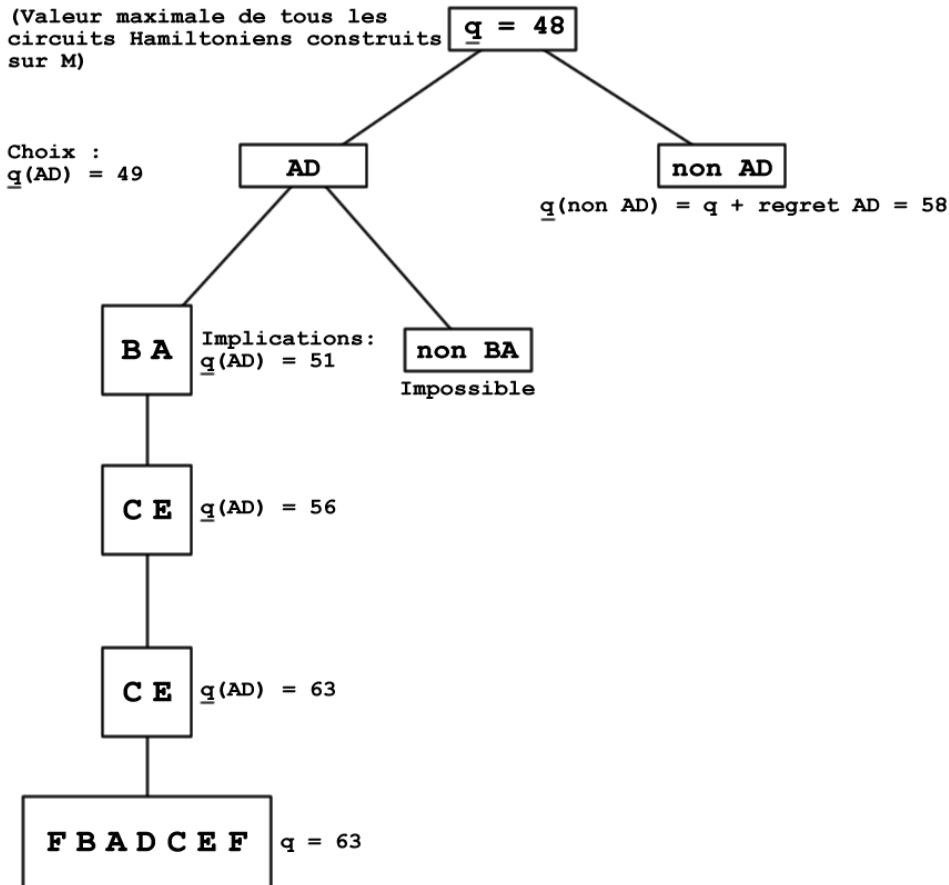


FIG. 5.32 – Résolution d'un problème du voyageur de commerce.

Nous n'avons aucune garantie quant à l'optimalité de ce circuit puisque l'arbre de recherche montre encore un rameau qui peut être développé avec un coût convenable. Il correspond à la négation de notre seule hypothèse; c'est le cas où le circuit ne passe pas par l'arc AD . Nous reconstruisons la matrice initiale dans ce cas de figure, en opérant la réduction prévue et nécessaire en ligne A . Mais le coût élevé de cette réduction entraîne l'élimination de nouveaux arcs : tous ceux dont le coût dépasse ou atteint la quantité $q^* - q(\text{non } AD) = 63 - 58$, soit 5 unités. Il reste le tableau :

	A	B	C	D	E	F
A	-	1	-	-	4	0
B	1	-	-	0	-	-
C	-	-	-	-	-	0
D	0	0	-	-	2	2
E	2	-	-	-	-	0
F	-	0	0	4	0	-

Ce tableau montre que CF (ligne C) et FC (colonne C) sont tous deux obligés. Cela n'est pas possible dans un circuit hamiltonien de plus de deux sommets, ce qui assure qu'il n'y a aucune solution meilleure que 63 unités sur cette branche et que le circuit précédent était optimal (Il est même possible à peu de frais de montrer que l'optimum obtenu était unique : il suffit de réintroduire dans la matrice précédente le seul arc de coût 5, l'arc CE , FC et CE sont alors forcés et impliquent de retenir aussi BD ; mais seuls les coûts nuls peuvent alors être encore pris en considération pour conserver à la

solution une valeur égale à 63 : or, retenir *DA* conduit à rejeter tout arc acceptable en colonne *B*). **Le seul circuit optimal pour notre voyageur de commerce correspond donc à un coût de 63, c'est ADCEFBA.**

5.13 Un programme général de résolution de problèmes

GPS - « General Problem Solver »- est le nom d'un célèbre programme conçu par Alan Newell, Cliff Shaw et Herbert Simon à la fin des années 50, puis repris par Georges Ernst et Newell en 1967. C'est l'un des tout premiers programmes d'intelligence artificielle capable de résoudre, dans une **démarche unifiée** - et non à l'aide d'un ensemble de procédures spéciales pour chaque cas - des problèmes aussi différents que : le calcul d'une intégrale, des casse-têtes logiques, la démonstration de théorèmes dans le calcul des prédicts, l'analyse grammaticale d'une phrase.

Les problèmes communiqués au programme GPS le sont toujours sous la forme : un objet initial : un objet final ; un ensemble d'opérateurs. Les *opérateurs* permettent d'effectuer des transformations sur les objets et d'atteindre ainsi à partir de l'objet initial, de proche en proche, l'objet final qui est le but désiré.

Remarquons que GPS est donc d'emblée dans une situation favorable par rapport à un étudiant devant sa feuille blanche, puisque ce dernier n'a pas explicitement, en général, la liste des opérations légales, mais doit au contraire la retrouver, ou même quelquefois l'inventer en cours de résolution.

5.13.1 La démarche générale de GPS : les sous-problèmes et les plans

GPS s'efforce continuellement de décomposer le problème posé en sous-problèmes plus simples selon la démarche descendante. Pour réaliser cette décomposition de façon adéquate, il utilise les différences existant entre l'objet en cours de traitement et l'objet désiré. Une différence entre deux objets est « toute caractéristique présente dans l'un des objets et absente ou altérée dans l'autre ». GPS choisit d'appliquer uniquement les opérations qui peuvent réduire la différence constatée à une étape donnée de la recherche. Sa démarche est ainsi très orientée, en fonction des moyens qui permettent d'arriver aux fins (« means-ends » analysis) et se rapproche à cet égard étroitement de la démarche humaine dans le même contexte.

Les buts. GPS connaît trois buts principaux qui sont :

- | | |
|------------------------|---|
| $A(O, x) =$ | Appliquer un opérateur <i>O</i> à un objet <i>x</i> . |
| $C(O, \Delta, x, y) =$ | Chercher un opérateur <i>O</i> qui puisse réduire une différence Δ entre les objets <i>x</i> et <i>y</i> . |
| $T(x, y) =$ | Transformer un objet <i>x</i> en un objet <i>y</i> . |

Ainsi, tous les problèmes comportent au départ le but premier :

$$T(\text{objet initial}, \text{objet final})$$

Les méthodes : Pour chacun de ces trois buts, le programme connaît les méthodes appropriées qui lui ont été données une fois pour toutes sous la forme :

Si le but est $A(O, a)$	ALORS comparer l'entrée de O avec l'objet a ; soit Δ la différence ; SI Δ est vide ALORS succès FIN. SINON créer le nouveau but : $C(Q, \Delta, a, b)$ et mettre le but $A(O, a)$ en attente.
Si le but est $C(O, \Delta, x, y)$	ALORS prendre les opérateurs les uns après les autres et voir s'ils peuvent réduire la différence Δ , SI oui ALORS créer le nouveau but $A(O, x)$. ALORS soit Δ la différence entre x et y . SI $\Delta = \text{vide}$ ALORS succès FIN. SINON créer les nouveaux buts : $C(Q, \Delta, x, y)$ et $A(Q, x)$. SI Q ne peut pas s'appliquer ALORS échec. SINON soit x' le nouvel objet obtenu, créer le nouveau but : $T(x', y)$.
Si le but est $T(x, y)$	

Pour améliorer les performances et bien que cela ne soit pas indispensable, les auteurs de GPS donnent de plus une indication sur l'importance relative (la difficulté pour réduire) des différences possibles. En outre, ils fournissent une table de connexion indiquant les opérateurs qui conviennent pour réduire chaque type des différences (voir les exemples qui suivent). A partir de l'objet initial et selon le premier des buts, GPS cherche donc un ou plusieurs buts convenables suivant les méthodes ci-dessus. A leur tour, ces buts sont développés grâce aux opérateurs qui leur sont applicables tant que l'objet final n'est pas rencontré et tant qu'il reste des objets qui n'ont pas encore été complètement développés.

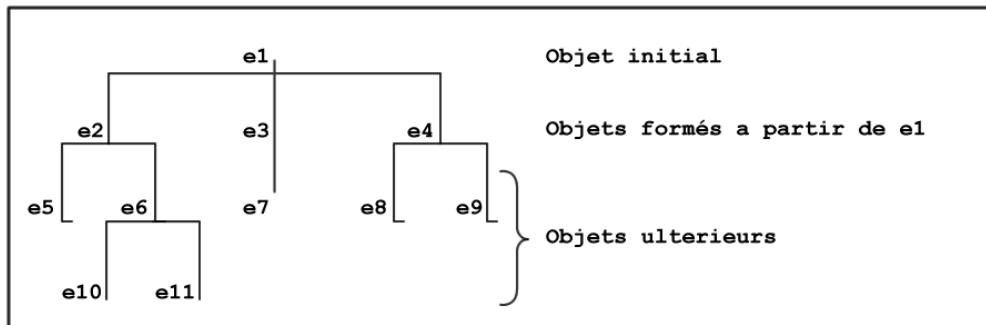


FIG. 5.33 – Arbre de recherche de GPS.

La démarche suivie se représente simplement à l'aide d'un schéma de type arborescent appelé « arbre de recherche ». A chaque instant, GPS choisit, dans cet arbre, le sommet pendant le plus prometteur, c'est-à-dire celui dont la différence avec l'objet final est la plus faible.

5.13.2 Un premier problème

Le problème du « singe et des bananes » a été proposé par John McCarthy en 1963. Il s'énonce : « Un singe et une caisse se tiennent dans une pièce au plafond de laquelle, suspendues à une ficelle, pendent des bananes. Le singe peut se déplacer, peut transporter la caisse ou monter dessus, il ne peut atteindre les bananes que s'il est sur la

caisse, celle-ci étant située juste sous les bananes ». Comment doit-il s'y prendre ? Ce problème, qui semble trivial pour un humain d'âge adulte, demande un petit effort de raisonnement qui va permettre de préciser complètement le comportement de GPS (Il a déjà été rencontré dans le chapitre « Applications de l'algorithme d'unification » et traité alors par le principe de Résolution).

1. Environnement

Partie A : divers éléments de l'univers : singe S , caisse C , bananes B .
 ensemble de places = $\{p_1, p_2, \text{sur} - C, \text{sous} - B, \dots\}$

Partie B : opérateurs O_1 .MONTER.condition : $\text{Place}(S) = \text{Place}(C)$
 action : $\text{Place}(S) = \text{sur} - C$
 O_2 .MARCHER.condition : La variable x est une place ;
 action : $\text{Place}(S) = x$
 O_3 .DEPLACER.condition : La variable x est une place ;
 $\text{Place}(S) \neq \text{sur} - C$
 $\text{Place}(S) = \text{Place}(C)$
 action : $\text{Place}(S) = x ;$
 $\text{Place}(C) = x$
 O_4 .ATTRAPER.condition : $\text{Place}(C) = \text{sous} - B$
 $\text{Place}(S) = \text{sur} - C$
 action : $\text{Contenu(main } S\text{)} = B ;$

Partie C : différences $D_1 = \text{Place}(S)$.
 $D_2 = \text{Place}(C)$.
 $D_3 = \text{Contenu(main } S\text{)}.$

Partie D : ordre des différences D_3 plus difficile que D_2 .
 D_2 plus difficile que D_1 .

Partie E : tables des connexions : (réduction des différences)

	D_1	D_2	D_3
O_1	×		
O_2	×		
O_3	×	×	
O_4			×

2. Tâche spécifique

But majeur \equiv Transformer l'objet initial en objet final.

Objet initial $\equiv [\text{Place}(S) = p_1, \text{Place}(C) = p_2, \text{Contenu(main } S\text{)} = \text{vide}]$

Objet final $\equiv [\text{Contenu(main } S\text{)} = B]$

Entrée du problème : GPS ne connaît, a priori, rien de chacun des problèmes qui lui sont soumis : il convient donc de lui communiquer, outre le problème lui-même, l'univers dans lequel il doit travailler. L'énoncé comporte ainsi deux parties : d'une part l'environnement général qui reste valable pour toute une famille de tâches, d'autre part la tâche précise demandée :

Commentaires et remarques :

Le tableau précédent est donné presque sous cette forme au programme ; il se comprend aisément. Les prédictats d'une variable : Place et Contenu, ont pour valeur respectivement la place et le contenu relatif à ladite variable. La partie D indique les difficultés relatives pour changer soit la place du singe S , soit celle de la caisse C , soit enfin, le contenu de la main du singe. Une croix dans la table de la partie E correspond à un opérateur a priori efficace pour réduire une différence de type donné. Comme il a été

déjà signalé, ces deux dernières parties de la description de l'environnement ne sont nullement indispensables : elles rendent simplement la recherche plus rapide et l'exposé moins long ...

Résolution : Le programme va s'efforcer de réduire successivement les différences entre l'objet initial, que nous noterons OB_i , et l'objet final, que nous noterons OB_f , en s'attaquant à elles dans l'ordre qui lui a été donné. Pour cela il construit des buts et des objets intermédiaires qui sont reliés les uns aux autres de façon naturelle par leurs relations de filiation suivant « un arbre généalogique » qui est organisé comme le montre la figure 5.34.

Dans cette figure les numéros associés aux sous-buts correspondent à l'ordre de leurs créations. Trois différences sont tous d'abord détectées entre OB_i et OB_f . L'objectif prioritaire est de réduire la différence D_3 , aussi le nouveau but créé est-il : but 2 = $C(O, D_3, OB_i, OB_f)$.

L'opérateur 4 est le seul à pouvoir réduire cette différence et un nouveau but est créé pour tenter de l'appliquer à l'objet initial : but 3 = $A(O_4, OB_i)$; mais ceci ne marche pas immédiatement car les deux conditions demandées par l'opérateur 4 ne sont pas satisfaites actuellement : la place de la caisse n'est pas correcte (différence D_2) et celle du singe ne l'est pas non plus (différence D_1). Le programme ajoute donc un autre but dans lequel la différence jugée la plus importante, à savoir D_2 , aura disparue ; c'est le but 4 = $C(O, D_2, OB_i, (Place(C) = sous - B))$ qui engendre à son tour le but 5 une fois trouvé l'opérateur convenable.

Bien entendu celui-ci ne s'applique encore pas immédiatement et la différence D_1 doit elle aussi être réduite : c'est le rôle des buts 6 et 7, qui imposent $Place(S) = Place(C)$. En 7, l'opérateur O_2 peut effectivement être déclenché. A partir de l'objet initial, il donne naissance à l'objet OB_1 ; OB_1 : $Place(S) = p_2$, $Place(C) = p_2$, $Contenu(main - S) = \text{vide}$.

Dès lors, le programme revient en arrière dans sa liste de buts, en notant que la différence D_1 a pu être réduite. Le premier but non rempli est donc maintenant le cinquième dans lequel il convient de substituer OB_1 à OB_i et le nouveau but $A(O_3, OB_i)$ introduit le nouvel objet OB_2 : OB_2 : $Place(S) = sous - B$, $Place(C) = sous - B$, $Contenu(main - S) = \text{vide}$.

De manière semblable, le retour en but 3 engendre trois autres sous-buts : but 9, 10 et 11. La différence D_1 est réduite en but 11 par le seul opérateur adéquat soit O_1 et finalement, en but 12, le singe se saisit des bananes.

Cet exemple a le mérite de montrer dans le détail comment un raisonnement peu complexe, un raisonnement « de tous les jours », a pu être complètement décomposé en une suite d'actions élémentaires.

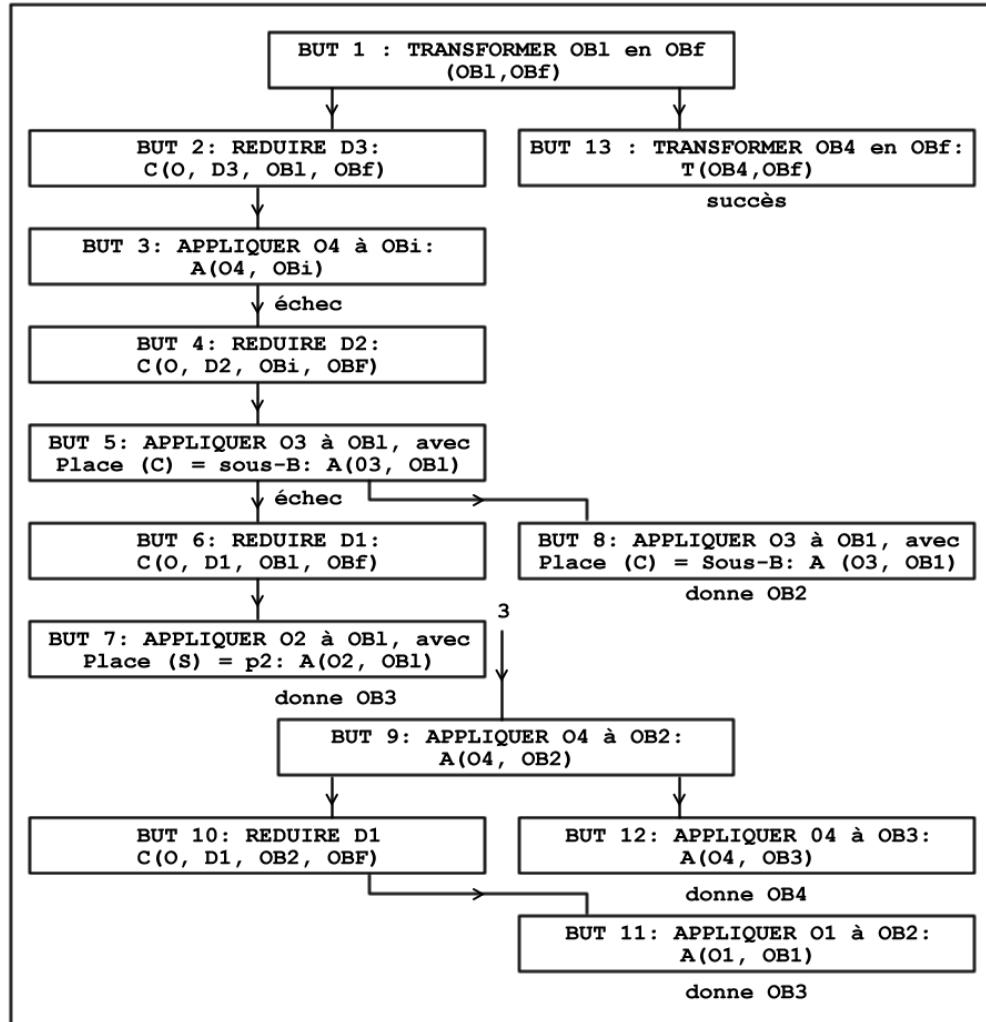
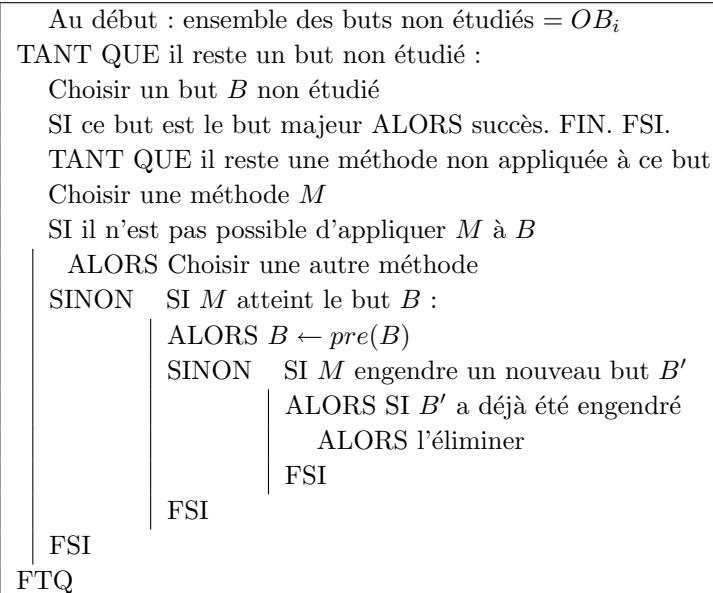


FIG. 5.34 – Arbre de recherche de GPS pour l'exemple

La « mécanique » qui permet d'atteindre le but est **remarquablement simple** et elle est facilement mise en œuvre. En outre, cette recherche a été efficacement guidée par l'introduction du **concept de différence** entre les objets de l'univers et l'utilisation de ces différences indésirables pour créer de manière naturelle de nouveaux buts à atteindre.

Bien sûr, il se peut que, dans un cas moins favorable, le programme s'engage dans un cul-de-sac, créant une suite de sous-buts en filiation directe, pour se rendre finalement compte que le but extrême ne peut jamais être atteint parce qu'aucun opérateur ne peut lui être appliqué. De plus, rien n'interdit que dans la recherche un même but intermédiaire réapparaisse en des endroits différents. Il est alors inutile de la conserver plus d'une fois mais il n'y a pas de risque de tourner en rond, en suivant la procédure générale suivante :



L'approche de GPS, résumée dans la procédure précédente, dans laquelle les choix se font à l'aide de la table de connexions et de l'ordre des différences, est tout à fait générale. Regardons maintenant ce programme opérer dans un contexte notamment différent puisqu'il s'agit de démonstration de théorèmes.

5.13.3 Un problème de logique formelle

L'univers de travail est ici la logique des propositions du premier ordre, mais aucune connaissance dans ce domaine n'est requise ; considérer l'exercice qui suit comme un simple jeu, où l'on doit manipuler des chaînes de caractères suivant des règles. D'ailleurs, à la vérité, avant d'être soumis à GPS, ce genre de problème avait été proposé par deux psychologues américains, O. Moore et S. Anderson (1954) à des étudiants pour tenter de comprendre comment l'homme agissait dans ce type de situation.

Rappelons qu'à cette époque l'algorithme d'unification n'était pas connu (cf chapitre 3 : Les systèmes formels).

Soient une chaîne de symboles de départ telle : $(R \supset \neg P) \wedge (\neg R \supset Q)$ et une seconde chaîne d'arrivée telle : $\neg(\neg Q \wedge P)$, il s'agit de trouver une séquence de transformations légales qui permette de passer de l'une à l'autre.

La table des transformations légales est :

r_0	$\neg\neg A \rightarrow A$
r_1	$A \wedge B \rightarrow B \wedge A$
	$A \supset B \rightarrow B \vee A$
r_2	$A \supset B \rightarrow \neg B \supset \neg A$
r_3	$A \wedge A \leftrightarrow A$
	$A \vee A \leftrightarrow A$
r_4	$A \wedge (B \wedge C) \leftrightarrow (A \wedge B) \wedge C$
	$A \vee (B \vee C) \leftrightarrow (A \vee B) \vee C$
r_5	$A \vee B \leftrightarrow \neg(\neg A \wedge \neg B)$
	$A \wedge B \leftrightarrow \neg(\neg A \vee \neg B)$
r_6	$A \supset B \leftrightarrow \neg A \vee B$
r_7	$A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C)$
	$A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$
r_8	$A \wedge B \rightarrow A$
	$A \wedge B \rightarrow B$
r_9	$A \rightarrow A \vee X$
r_{10}	$\begin{array}{c} A \\ B \end{array} \rightarrow A \wedge B$
r_{11}	$\begin{array}{c} A \\ A \supset B \end{array} \rightarrow B$
r_{12}	$\begin{array}{c} A \supset B \\ B \supset C \end{array} \rightarrow A \supset C$

Les symboles r_1 à r_{11} sont de simples repères qui ne font pas partie des règles.

Le symbole \rightarrow situé entre deux chaînes de symboles C_1 et C_2 se lit « peut se réécrire » ; il signifie que l'on a le droit de transformer à tout instant C_1 en C_2 . La double flèche \leftrightarrow signifie quant à elle à la fois $C_1 \rightarrow C_2$ et $C_2 \rightarrow C_1$.

Lors de toute réécriture il est, de plus, permis de **substituer** à toute lettre apparaissant dans la règle considérée toute autre lettre, sous réserve de faire la même substitution de part et d'autre du signe \rightarrow .

Une lettre précédée du symbole \neg peut se substituer à une lettre quelconque.

Tous les symboles qui ne sont pas des lettres **ne sont pas substituables** par hypothèse.

Finalement, une règle sera effectivement applicable à une chaîne lorsque, à la suite de substitutions, on sera parvenu à **faire rigoureusement coïncider** symbole après symbole, **la partie gauche de la règle**, i.e. partie qui précède le signe \rightarrow , et **une partie de la chaîne étudiée**. Les lettres jouent, on le voit, dans ce formalisme le rôle des variables dans le calcul classique des propositions ; une substitution est ici un simple changement de variable.

Ainsi, partant de la chaîne :

$$(P \supset Q) \wedge S$$

Je peux appliquer la règle r_2 en substituant P à A et Q à B pour obtenir :

$$(\neg Q \supset \neg P) \wedge S'$$

Je peux maintenant appliquer la règle r_6 en substituant $\neg Q$ à A et $\neg P$ à B , pour transformer l'expression en $(\neg\neg Q \vee \neg P) \wedge S'$.

Une dernière réserve subsiste enfin pour les règles r_8 à r_{12} : contrairement aux huit premières règles, celles-ci ne permettent de transformer que des chaînes prises dans leur ensemble. Les parties gauches de ces règles, dites règles de production (cf chapitre 3), doivent venir coïncider avec des chaînes complètes, i.e. non strictement situées à l'intérieur d'autres chaînes. Ainsi, r_8 permet de réécrire :

$$(P \vee Q) \wedge R \text{ en } P \vee Q \text{ et en } R,$$

mais ne s'applique pas à :

$$(P \wedge Q) \vee R$$

Moyennant les conventions de substitutions précédentes, la table donnée des règles correspond absolument à la table des opérateurs dans le problème du singe et des bananes : comme le faisait un opérateur, une règle transforme, suivant une certaine loi et sous réserve de certaines conditions, un objet de l'univers en un autre. Nous pouvons à nouveau dresser un tableau indiquant quelles règles conviennent pour réduire telle ou telle différence.

La nature des différences possibles entre les objets est ici plus variée. Deux chaînes quelconques peuvent en effet présenter :

- 1) des nombres de lettres différents : augmentant (D_1) ou diminuant (D_2) ;
- 2) des nombres de symboles différents (D_3) ;
- 3) des nombres d'autres connectives différents (D_4) ;
- 4) un changement de parenthésage (D_5) ;
- 5) une transposition dans l'ordre des lettres (D_6).

L'observation des effets respectifs des règles données sur ces différences permet de dresser la table de connexion : (opérateurs × différences) :

Opérateurs Différences	r_0	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}	r_{12}
D_1 addition d'une lettre				×				×		×	×		
D_2 soustraction d'une lettre				×				×	×			×	×
D_3 modification du nombre de symboles \neg	×		×			×	×						
D_4 modification d'une autre connective \neg						×	×	×					
D_5 parenthé- sage \neg					×				×				
D_6 transposi- tion dans l'ordre des lettres \neg		×	×										

FIG. 5.35 – Table de connexions pour GPS dans le problème de logique

Nous supposerons, cette fois, que nous n'indiquons aucun ordre sur les difficultés relatives des différences. GPS peut maintenant travailler en partant de :

$$OB_i : \text{ objet initial } \equiv (R \supset \neg P) \wedge (\neg R \supset Q) \text{ et de}$$

$$OB_f : \text{ objet final } \equiv \neg(\neg Q \wedge P).$$

Résolution : La première différence constatée par le programme entre ces deux objets est la disparition dans OB_f de la « variable » R .

Le premier but engendré est donc REDUIRE D_2 avec en paramètre la variable R . Suivant les indications notées une fois pour toutes dans la table de connexions, les règles $r_3(\rightarrow)$, $r_7(\leftarrow)$, r_8 , r_{11} et r_{12} peuvent permettre d'atteindre cet objectif. Seule parmi

celles-ci, la règle r_8 s'applique directement, d'abord sur la partie gauche de OB_i , pour donner naissance à un nouvel objet : $OB_1 \equiv R \supset \neg P$.

(Notons ici que, contrairement à ce qui se passe dans l'univers habituel et, en particulier, pour l'univers du premier problème, en démonstration de théorèmes l'application d'une règle **ne détruit pas** l'expression de départ ; la transformation ajoute seulement le nouvel objet à l'univers. Ceci facilite le travail puisqu'il n'y a plus besoin de retour-arrière pour revenir à un état déjà obtenu : **tous les objets restent ici présents en même temps**, alors que le singe ne pouvait pas occuper à la fois deux places distinctes.)

Les membres gauches des règles $r_3(\rightarrow)$, $r_7(\leftarrow)$, r_{11} et r_{12} ne peuvent pas pour l'instant coïncider avec la chaîne OB_i . Les différences sont donc notées et autant de sous-but de type $C(O, \Delta, OB_i, \text{ membre gauche de } r_j)$ avec $j = 3, 7, 11$ et 12 , sont ajoutés. En outre, GPS note que la règle r_8 peut être appliquée à OB_i d'une autre manière.

La stratégie générale de GPS pour choisir le but duquel il va repartir à l'étape suivante consiste à **développer le but le plus profond** dans l'arbre de la recherche : c'est ici OB_1 . Comparé à OB_f , OB_1 montre une différence D_1 et une différence D_2 , puisqu'en fait ce n'est pas la bonne lettre R qui a disparue en passant de OB_i à OB_1 , mais la lettre Q .

Mais GPS juge le sous-but correspondant à la réduction simultanée de ces **deux** différences, **plus difficile** que le but primitif. Aussi ce sous-but est-il abandonné au profit du but le plus profond, qui n'a pas encore été développé : cela correspond à l'application de la règle 8 à la partie droite de OB_i . Cependant R ne disparaît toujours pas et la recherche se poursuit avec OB_i et r_7 .

Le début de la recherche de GPS sur ce problème est ainsi dans l'ordre réel de traitement, avec entre crochets quelques commentaires, celui du schéma suivant :

But 1 : TRANSFORMER OB_i en OB_f

But 2 : REDUIRE D_2 (Variable R)

But 3 : APPLIQUER r_8 à gauche de OB_i
donne $OB_1 \equiv R \supset \neg P$

But 4 : TRANSFORMER OB_1 en OB_f

But 5 : REDUIRE D_2 (variable Q)

But 6 : REDUIRE D_1 (variable R)

RETOUR :

(Coût trop fort par rapport à But 1)
(Relance du But 2)

But 2 :

But 7 : APPLIQUER r_7 à droite de OB_i
donne $OB_2 \equiv \neg R \supset Q$

But 8 : TRANSFORMER OB_2 en OB_f

But 9 : REDUIRE D_2 (variable P)

REDUIRE D_1 (variable R)

RETOUR :

(Coût trop fort par rapport à But 1)

But 2 :

But 10 : APPLIQUER $r_7(\leftarrow)$ à OB_i
(impossible, d'où génération
d'un nouveau but)

But 11 : REDUIRE D_4 à gauche de OB_1 (connective V)

But 12 : APPLIQUER r_6 à OB_i

(car la réduction précédente n'est
pas directement possible)
donne $OB_3 \equiv (\neg R \vee P) \wedge (\neg R \supset Q)$

But 13 : APPLIQUER $r_7(\leftarrow)$ à OB_3 (nouvel essai)

But 14 : REDUIRE D_4 à droite de OB_4 (connective V)

But 15 : APPLIQUER r_6 à OB_3

donne $OB_4 \equiv (\neg R \vee \neg P) \wedge (\neg \neg R \vee Q)$

But 16 : APPLIQUER $r_7(\leftarrow)$ à OB_4 (nouvel essai)

But 17 : REDUIRE D_3 à gauche de la droite de OB_4

But 18 : APPLIQUER r_6 à droite de OB_4

RETOUR :

(on retrouve OB_3)

(retour en But 2)

But 2 :

But 19 : APPLIQUER r_3 à $OB_i \dots$

GPS poursuit sa recherche... Au total il va engendrer 52 buts avant de parvenir à :

But 53 : APPLIQUER r_{12} à OB_2 : échec
 condition : trouver $OB_{10} \equiv A \supset \neg R$

But 54 : TRANSFORMER OB_i en OB_{10}

But 54 : TRANSFORMER OB_1 en OB_{10}
 (But équivalent et jugé plus facile)

But 56 : REDUIRE D_3

But 56 : APPLIQUER r_2 à OB_1
 donne $OB_{11} = \neg\neg P \supset \neg R$

But 53 (relancé) :
 APPLIQUER r_{12} à OB_{11} et OB_2
 donne $OB_{12} \equiv \neg\neg P \supset Q$

But 8 (relancé) :
 TRANSFORMER OB_{12} en OB_f

But 58 : REDUIRE D_3 (variable P)

But 59 : APPLIQUER r_0 à OB_{12}
 donne $OB_{13} \equiv P \supset Q$

But 60 : REDUIRE D_4 (connective \supset)

But 61 : APPLIQUER $r_6(\rightarrow)$ à OB_{13}
 donne $OB_{14} \equiv \neg P \vee Q$

But 62 : REDUIRE D_4 (connective \vee)

But 63 : APPLIQUER $r_5(\rightarrow)$ à OB_{14}
 donne $OB_{15} \equiv \neg(\neg\neg P \vee \neg Q)$

But 64 : REDUIRE D_3

But 65 : APPLIQUER r_0 à OB_{15}
 donne OB_{15}

But 66 : REDUIRE D_6

But 67 : APPLIQUER r_1
 donne $OB_{17} \equiv \neg(\neg Q \wedge P)$
 (Succès) FIN

A la lumière de ces deux exemples, il est clair que la méthode manuelle, pour résoudre ce genre de problèmes, n'est pas fondamentalement différente de celle de GPS. L'idée de raisonner en termes de différences et de caractériser les opérateurs légaux pour les différences qu'ils sont capables de réduire est **une bonne idée générale**.

Bien sûr, nous avons souvent une bonne perception de l'univers de travail ce qui nous permet d'être plus direct et moins lourd que GPS quelquefois. Mais nous **abandonnons** aussi peut-être plus vite là où GPS s'obstine et finit par arriver !

La figure 5.36 donne l'arbre de cette recherche.

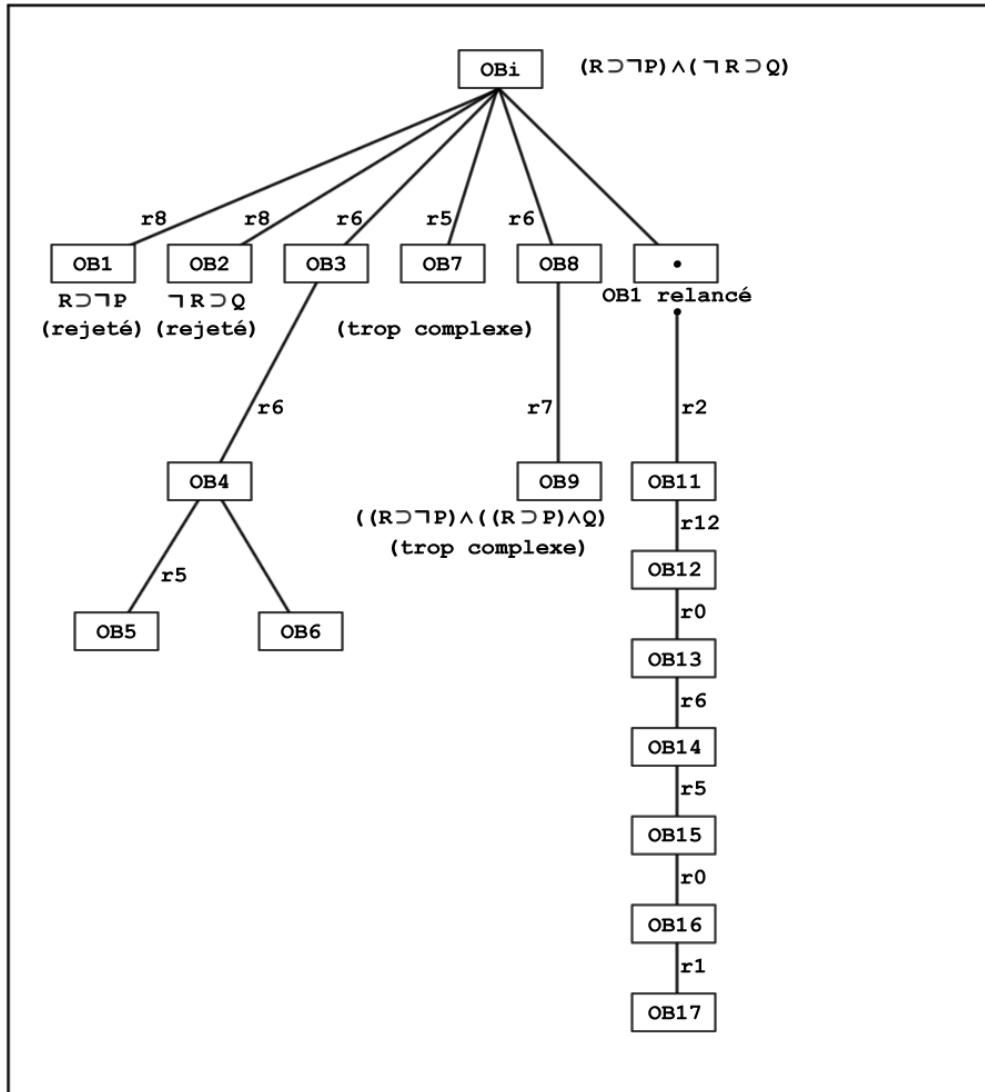


FIG. 5.36 – Arbre de recherche de GPS en logique formelle

5.13.4 Autres tâches données à GPS

Dans sa dernière version, due à Ernst et Newell (1967), GPS a travaillé sur onze tâches différentes. Un point d'information intéressant est que le programme demandait environ 20 secondes par but engendré. Ainsi le problème du singe et des bananes qui, nous l'avons vu, utilisait 13 buts prenait presque quatre minutes sur l'IBM 7090 de l'Université de Carnegie-Mellon.

Intégration formelle : la première tâche donnée dans ce nouvel univers fut « intégrer $te^{t^2} dt$ ». Le résultat $\frac{e^{t^2}}{2}$ est obtenu en trois minutes après créations de 11 buts. L'expression $(\sin^2 \alpha t \cos \alpha t + t^{-1})dt$ est intégrée en $\log t + (3\alpha)^{-1} \sin^3 \alpha t$ moyennant un temps et un nombre de buts à peu près double.

Il faut remarquer que dans cet univers, l'objet final n'est pas aussi bien défini qu'à l'accoutumée. Ici, à la vérité, l'objet final doit seulement vérifier la propriété : ne pas contenir le symbole d'intégration.

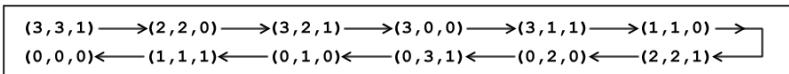
Ceci ne rend pas ce type de problème plus difficile, mais ne fait qu'introduire une priorité absolue sur les différences avec la disparition du signe d'intégration en tête de liste.

Six opérateurs, bien adaptés à **six cas d'intégration** standard, sont ici donnés à GPS en même temps que quatre autres opérateurs de différenciation. Un dernier opérateur signale que l'intégrale d'une somme est la somme des intégrales. De plus, tous les calculs algébriques classiques : addition, multiplication, associativité et commutativité, sont programmés spécialement et incorporés au programme GPS lui-même.

Les missionnaires et les cannibales : « Trois missionnaires et trois cannibales se trouvent ensemble d'un côté d'une rivière que tous désirent traverser. Le seul moyen de transport disponible est une barque qui peut au plus supporter le poids de deux personnes. En outre, si à un moment quelconque, le nombre de cannibales dépasse le nombre de missionnaires, ces derniers sont impitoyablement dévorés par les premiers. que cela se produise sur une rive ou sur l'autre. Comment faire traverser tout le monde, et la barque, de l'autre côté de la rivière ? »

Un objet est ici constitué de deux parties correspondant chacune à une rive. Chaque partie décrit le nombre de missionnaires, de cannibales et la présence ou l'absence du bateau. Il y a alors cinq opérateurs de transformation pour chacun des deux sens de traverse, puisque soit un cannibale, soit un missionnaire, soit un cannibale et un missionnaire, soit deux cannibales, soit enfin deux missionnaires peuvent se trouver dans la barque.

Tout ceci conduit GPS dans son approche moyens/fins, aisément simulable manuellement, à construire 57 buts pour parvenir finalement à la solution :



Dans ce schéma, on a retenu la description de la rive de départ sous la forme : (nombre de missionnaires, nombre de cannibales, présence (= 1) ou absence (= 0) du bateau). Il est curieux de constater que cette solution présente une symétrie par complément à (3, 3, 1) après la sixième traversée. En fait, cette propriété s'explique et est conservée pour d'autres exemples du même problème comme l'a montré Saul Amarel (1968).

Les tours de Hanoï : Ce problème a été inventé de toutes pièces par le mathématicien français Edouard Lucas. Il s'agit d'un casse-tête où l'on doit déplacer des disques d'un piquet à un autre. « Dans la situation initiale, on a trois piquets, A, B et C et quatre disques de tailles strictement décroissantes enfilés sur le piquet A, le plus petit disque en haut. Un coup consiste à déplacer un disque du sommet d'un piquet au sommet d'un autre, mais, à aucun moment, un disque ne doit être posé sur un disque plus petit que lui. Comment transporter les quatre disques de A en C ? »

Comme pour tous les problèmes précédents, certaines conditions doivent donc être remplies pour pouvoir appliquer effectivement un des six opérateurs possibles ; mais, parce que dans cet univers simple la méthode des différences est particulièrement adaptée, GPS va droit à la solution en 46 étapes **sans engendrer aucun but inutile**.

Les trois pièces : « Trois pièces de monnaie sont sur une table. Les deux pièces aux extrémités présentent leur côté face, tandis que celle du milieu présente son côté pile. Chaque coup consiste à retourner deux pièces quelconques. Le problème est en trois coups *exactement*, de faire venir toutes les pièces soit sur pile soit sur face ». Un compteur est spécialement ajouté aux descriptions des objets pour prendre en compte cette contrainte. GPS engendre seulement 10 buts pour mener cette tâche à bien.

Les sept ponts de Königsberg : « A Königsberg (aujourd'hui Presburg) la rivière Pregel sépare deux îles du continent. Sept ponts relient ces îles entre elles et au reste

de la ville comme l'indique la figure 5.37 ».

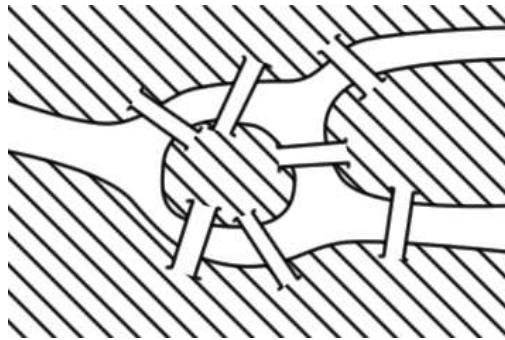


FIG. 5.37 – Les sept ponts de Königsberg

Les opérateurs correspondent dans ce problème aux quatorze traversées différentes possibles. Le problème est de « partir d'un point quelconque et d'y revenir en ayant traversé chaque pont *exactement une fois* ».

Ce problème est impossible, comme l'a montré Léonard Euler en 1736. Comme GPS n'a pas de méthode spécifique, il échoue dans cette tâche. Sa seule chance est d'épuiser toutes les possibilités mais après avoir trouvé plusieurs manières de franchir six ponts, il a saturé la mémoire de l'ordinateur.

Les transvasements : « Vous disposez d'une réserve d'eau illimitée, d'une cruche de 5 litres, d'une autre de 8 litres, comment obtenir 2 litres dans la petite cruche ? » GPS règle la question en 24 buts, après quelques tâtonnements, car il y a une grande différence entre les deux situations extrêmes.

Le père et ses fils : « Un père, ses deux fils et une barque sont d'un même côté d'une rivière : le père pèse 80 Kilos, chaque fils en pèse 40 et le bateau ne supporte que 80 kilos : comment faire traverser toute cette petite famille ? »

GPS répond en 33 buts.

Série de lettres à compléter : Il s'agit d' « ajouter les deux lettres suivantes à la série BCBDBE »...

Ce problème, comme les autres tests du même type, pose la question fondamentale de la mesure de la simplicité du résultat. Ici, on peut admettre qu'aussi bien BF que BC conviennent : pour la réponse BC, l'idée est de recommencer la série au début : bien que BF soit, justement, préférée en principe, la réponse n'est pas claire en général. Aussi, la façon dont le problème est transcrit pour GPS est ici particulièrement importante. Cette transcription est nettement moins directe que dans les autres cas précédemment présentés. Les objets donnés ne sont pas des séries de lettres mais des séries *partielles* avec certaines relations entre les lettres. Une relation entre deux lettres est soit MEME, qui marque l'identité, soit SUIT, qui indique que la seconde suit la première dans l'alphabet.

L'objet final, quant à lui, est simplement défini par trois conditions qui doivent être satisfaites une fois l'objet initial transformé : la première est que les blancs doivent avoir été remplacés (les deux blancs sont changés en deux lettres spéciales x et y) : la seconde est que l'ensemble des relations qui permettent de décrire la chaîne finale doit être le plus simple possible : si deux descriptions ont les mêmes relations, la plus simple est celle dont les opérandes sont séparées par le minimum de lettres de la série. La troisième condition est que toute lettre doit faire partie d'au moins une relation.

Deux opérateurs peuvent essentiellement être appliqués : l'un correspondant à assigner une relation à deux lettres de la série, l'autre à assigner une valeur à une lettre. En outre, un troisième opérateur oblige à parcourir la série de gauche à droite. Ainsi, GPS découvre et marque à mesure les propriétés rencontrées dans la série. GPS ne tâtonne pas en affectant a priori des valeurs à x et à y , mais, bien au contraire, il dé-

duit logiquement les valeurs qu'elles doivent prendre pour vérifier elles aussi les *mêmes* relations que les lettres avec lesquelles elles correspondent. L'étude de l'objet initial conduit alors, pour GPS, à la description donnée par la figure 5.38.

GPS détecte en parcourant la séquence de gauche à droite la même relation (MEME) entre le premier et le troisième et entre le cinquième et le troisième. Il impose alors que cette relation soit vérifiée encore ensuite et prend donc $x = B$. La partie complémentaire de la séquence est traitée ensuite parallèlement. Au total la séquence finale correcte est obtenue par cette démarche inductive en 27 buts.

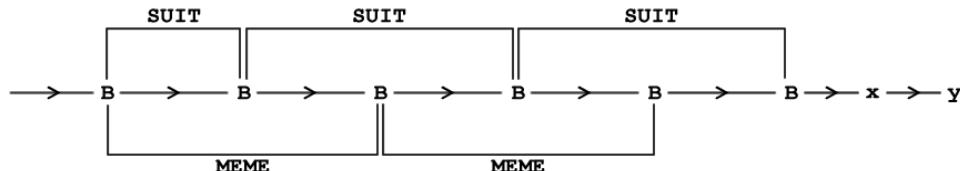


FIG. 5.38 – Séquence de lettres à compléter.

Analyse grammaticale : Il s'agit de faire l'analyse d'une phrase du langage courant suivant une grammaire donnée. Parmi les règles grammaticales données à GPS on trouve ici :

- r_1 : Une phrase est constituée d'un groupe nominal suivi d'un groupe verbal lui-même suivi d'un autre groupe nominal.
- r_2 : Un groupe nominal peut être un groupe adjectival suivi par un mot qui peut être utilisé comme un nom.
- r_3 : Un groupe nominal peut être un mot qui peut être utilisé comme un nom.
- r_4 : Un groupe adjectival est un mot utilisé comme adjectif.

Une autre règle spécifie que le mot « porte », par exemple, sera tantôt un nom tantôt un verbe ; « fort » sera tantôt un adjectif tantôt un nom.

Le but majeur est alors de reconnaître la suite de symboles : « une épaisse porte ferme le fort » (objet initial), comme une phrase (objet final). Puisque « épaisse » n'appartient qu'à la classe adjectif, GPS infère que « porte » est utilisé comme un nom, et suppose que « ferme » est un verbe, alors « fort » est un nom.

La phrase anglaise d'origine : « free variable cause confusion » est analysée en 19 buts. Ce problème montre l'intérêt d'une démarche de type heuristique dans des algorithmes comme ceux utilisés en analyse syntaxique.

Démonstration de théorèmes : Outre les tâches déjà mentionnées en calcul propositionnel, GPS est capable de prouver des théorèmes dans le calcul des prédictats du premier ordre en utilisant la technique particulière dite de « résolution », inventée en 1931 par Jacques Herbrand et programmée notamment par J.A. Robinson (1965) (Cf. chapitre 3). Le théorème classique Church (1965) :

$$\exists u \exists y, \forall z [P(u, y) \supset (P(y, z) \wedge P(z, z)) \wedge (P(u, y) \wedge Q(u, y))] \supset [Q(u, z) \wedge Q(z, z)]$$

est soumis à GPS après skolémisation et mise sous forme requise pour la Résolution (forme conjonctive normalisée de tous les prédictats sans quantificateurs). Le 59^{me} but permet d'exhiber une contradiction et achève donc la démonstration.

5.13.5 Conclusion

Ainsi malgré une certaine lenteur d'exécution et un nombre total de buts intermédiaires quelquefois trop important, GPS démontre la possibilité de concevoir un **système unique**, fondé sur une stratégie liée à la **détection** de **différences** et à l'utilisation locale de moyens disponibles pour **réduire** celles-ci, qui permet de résoudre une famille diversifiée de problèmes non triviaux. Ces idées furent reprises un peu plus tard et les performances nettement améliorées en utilisant des tables et des ordres de différences précis et bien adaptés dans deux domaines : l'intégration formelle d'une part, c'est le programme Saint de J. Slagle (1963), la logique, d'autre part avec J. Quinlan (1969). Cependant, alors que la question des performances n'est sans doute que peu importante, il reste trois reproches plus graves contre GPS.

En premier lieu, l'ensemble des problèmes posés n'est pas très représentatif ; ils sont tous un peu de la *même* famille et aucun n'est vraiment difficile. Si GPS réussit c'est donc, également, parce que sachant a priori qu'il traite ce type de problèmes, il sait aussi qu'il peut dans tous les cas réduire l'espace de recherche à très peu de chose.

En second lieu, il apparaît que la méthode des différences ne convient que si aucune vision *globale* de la tâche à accomplir n'est nécessaire dans la résolution.

C'est une méthode à courte vue, qui suppose qu'il existe une certaine continuité dans le chemin qui *mène* à la solution, qu'aucun large détour n'est en particulier nécessaire, que le problème peut se décomposer en une suite d'étapes élémentaires, toutes sensiblement égales en importance. Il existe bien sûr une grande famille de problèmes, même parmi les casse-têtes logiques ou les tâches formelles semblables à ceux soumis à GPS, où c'est loin d'être le cas. Nous en rencontrerons maints exemples dans la suite (Chapitres 6, 7 et 8).

Finalement en troisième lieu, et c'est là à mon sens le reproche principal, il n'est pas facile d'énoncer un problème pour le soumettre à GPS. En effet, l'**énoncé préjuge déjà de la résolution** : il faut communiquer au système l'ensemble convenable des différences dans chaque cas, la table de connexion et d'autres informations encore. En outre, il faut faire un effort intellectuel notable pour pouvoir décrire certaines situations en termes d'opérateurs ; ainsi en est-il des 7 ponts de Königsberg et des séries de lettres à compléter. Mais la conception de ce programme. Il y a plus de vingt-cinq ans, reste tout à fait admirable.

Chapitre 6

Les programmes de jeux ; les études des psychologues

Les jeux de société placent les joueurs dans des situations compétitives où chacun, à tour de rôle, doit prendre une décision sans connaître celle que prendra ensuite l'adversaire. Les êtres humains, pour résoudre ce problème, raisonnent classiquement en développant à l'avance un certain nombre de coups aussi bien amis qu'ennemis : si je joue un tel coup, alors l'adversaire peut répondre tel ou tel autre coup ; après l'un de ceux-ci, j'ai telle et telle riposte à ma disposition et ainsi de suite. Une arborescence des coups légaux, et donc des situations possibles, est ainsi engendrée (figure 6.1), où les coups amis et ennemis alternent à chaque changement de niveau.

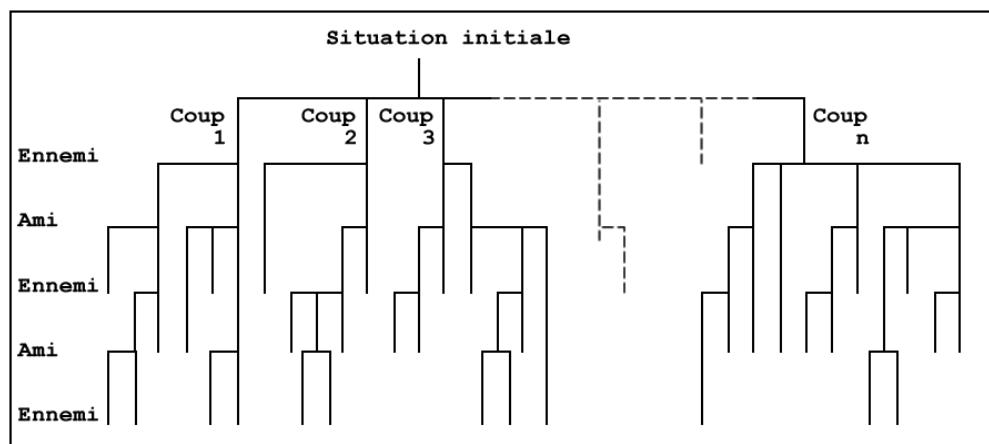


FIG. 6.1 – Exemple d'arborescence liée au raisonnement pour jouer un coup au jeu d'échecs.

Après un certain temps de réflexion, le joueur humain, pesant le pour et le contre des différentes situations, se décide pour le coup du premier niveau qu'il juge le meilleur et il le joue. Les programmes de jeux sur ordinateurs ne procèdent pas autrement. Ce chapitre a pour objet de décrire la méthode la plus communément employée dans ce but.

Nous étudierons tout d'abord la façon dont un tel programme construit son arborescence de recherche et l'intérêt de cette dernière. Ensuite, il s'agit de juger de la

valeur des positions obtenues. Ceci se fait à l'aide de ce qu'on appelle depuis 1950, avec Claude Shannon, une *fonction d'évaluation*, objet du second paragraphe. Enfin, le coup à jouer après ces développements et ces calculs doit être choisi : le troisième et le quatrième paragraphe décrivent la méthode de base et une amélioration de celle-ci qui permettent ce choix. Nous verrons alors combien tout ceci est proche des procédures que nous avons étudiées pour résoudre précédemment les problèmes des graphes. Finalement, nous discuterons cette approche pour en étudier les défauts et les limites et évoquerons la possibilité d'une méthode radicalement différente qui sera reprise et développée plus tard au paragraphe 6.6.8.

6.1 L'arborescence des coups légaux

L'arborescence n'est tout d'abord nécessaire que parce que nous ne savons pas évaluer correctement la situation initiale. Ainsi, au jeu d'échecs, personne ne peut dire au départ qui des blancs ou des noirs gagnerait si chaque côté jouait parfaitement. Le développement de l'arborescence est donc utile pour pallier ce défaut de l'évaluation. L'idéal serait alors d'envisager un ensemble de coups assez grand pour qu'on obtienne des situations, en fin de développement, qui soient évaluables exactement : un des rois est échec et mat, un des camps peut faire échec perpétuel, un des rois est pat. Malheureusement, il n'en est pas ainsi dans la plupart des positions.

En début de partie, aux échecs, chaque camp possède vingt coups à sa disposition : après le premier échange, quatre cents positions différentes sont déjà possibles et aucune ne permet, bien sûr, une évaluation définitive. En continuant à considérer tous les coups de part et d'autre, le nombre de coups moyen augmente pour atteindre cinquante vers le milieu de la partie. Si l'on décide donc d'étudier à l'avance un nombre d'échanges, l'arborescence engendrée présentera un nombre d'extrémités terminales ou feuilles, égal à : K^n . On a supposé ici le nombre de coups légaux à chaque niveau, pour chaque niveau et pour chaque joueur, constant et égal à K . En pratique, on peut poser aux échecs $K = 40$, d'où, pour par exemple 6 mouvements, un nombre de feuilles qui vaut :

$$40^6 = 4096000000$$

Ce nombre est considérable et chaque niveau a pour effet de le multiplier par le facteur K : la croissance de la recherche et en particulier celle du temps de réflexion nécessaire pour engendrer les coups et pour évaluer les situations terminales, est grossièrement proportionnel à $K \exp n$. Elle croît donc exponentiellement avec la profondeur. Cette explosion combinatoire interdit donc tout développement jusqu'à ces profondeurs suffisamment grandes pour donner des situations évaluables de façon sûre : un ordinateur ultra puissant qui pourrait engendrer et évaluer dix situations terminales chaque seconde et qui travaillerait ainsi depuis le temps estimé de formation de notre système solaire, soit 4,6 milliards de nos années, aurait « seulement » pu examiner un nombre de positions valant :

$$10 \times 4,6 \times 10^9 \times 366 \times 24 \times 3600$$

soit $15 \cdot 10^{18}$ ou encore environ 40^{12} donc 12 mouvements, c'est-à-dire 6 échanges. Ce qui est d'un ordre de grandeur double de celui trouvé précédemment pour une profondeur 6. Un tel ordinateur n'aurait pas même tout à fait pu examiner une douzaine de coups d'avance ! Quand on sait qu'une partie d'échecs dure le plus souvent une bonne quarantaine de coups, on comprend l'intérêt et la fascination de ce jeu chez les hommes !

C'est d'ailleurs pour les mêmes raisons, à savoir trouver des méthodes pour lutter contre l'explosion combinatoire, que les jeux, et en particulier le jeu d'échecs, ont souvent servi de support aux recherches en intelligence artificielle. Mais ici le but n'est pas tant d'avoir un programme qui joue bien que de découvrir les méthodes générales elles-mêmes, qui seront transposables dans d'autres domaines et permettront de résoudre d'autres problèmes. Un certain nombre de jeux sont ainsi intéressants du point de vue de l'intelligence artificielle ; ce sont tous les jeux pour lesquels l'arborescence précédente est assez petite pour être complètement étudiée par les ordinateurs actuels convenablement programmés, il en est ainsi par exemple du tic-tac-toe ou des problèmes d'échecs en deux ou trois coups ou encore de certains jeux, dits jeux de Nim, pour lesquels on peut construire une fonction d'évaluation parfaite de la position. Le jeu des allumettes popularisé par le film d'Alain Resnais « L'Année dernière à Marlenbad » entre dans cette famille.

Ainsi encore, le jeu de « tic-tac-toe » se joue sur un terrain de neuf cases reliées comme l'indique la figure 6.2. Chaque joueur dispose de trois pierres. Dans un premier temps, chaque joueur, à tour de rôle, pose une de ses trois pierres où il veut sur une case libre. Dans le second temps, quand les six pierres sont posées, chaque joueur, à son tour, déplace une de ses pierres au choix dans une des cases libres voisines (cases vides situées à un pas de distance en suivant les liaisons figurées sur le terrain). Le gagnant est celui qui le premier parvient à aligner ses trois pierres.

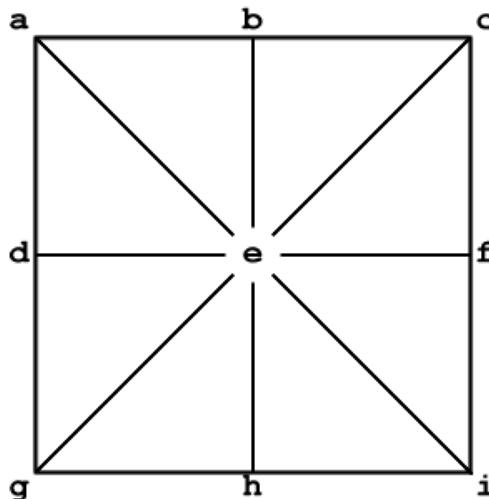


FIG. 6.2 – Le terrain au jeu de tic-tac-toe

Insistons bien ici sur le fait que la génération des coups légaux, définis par les règles du jeu considéré, ne pose jamais de problèmes. Elle est toujours de nature algorithmique. Donnons le schéma de principe de cet algorithme pour les (nombreux) jeux à deux joueurs où les pièces se déplacent sur un échiquier. Considérant l'une après l'autre les cases de l'échiquier, on ne s'intéressera qu'à celles qui sont occupées par une pièce du camp dont on veut engendrer les coups. Pour chacune d'elles, on envisagera les déplacements dans chacune des directions autorisées par les règles. Soient alors pour chaque case (i, j) :

Occupation (i, j) :	La nature actuelle de la case. Elle peut être vide, amie ou ennemie.
Pièce (i, j) :	La nature de la pièce sur la case dans le cas où $\text{occupation}(i, j) \neq \text{vide}$.
Soient, en outre, n directions	Le nombre total de directions de déplacement possibles dans le jeu considéré
Directions autorisées	Celles de ces directions qui sont autorisées pour la pièce p .

Ainsi, au tic tac toe, il y a, a priori, 8 directions possibles :

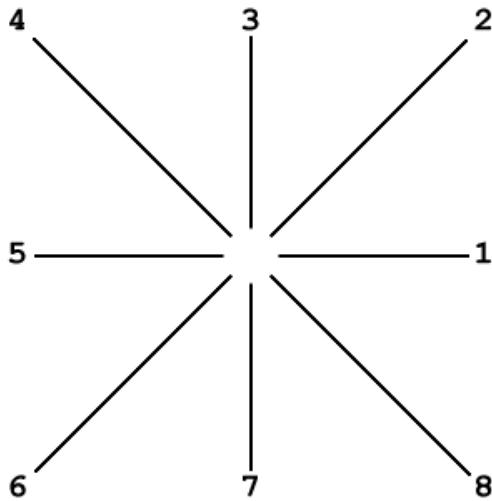


FIG. 6.3 – Directions possibles au tic tac toe (ou aux échecs)

Toutes les pièces sont de même nature et toutes les directions sont en principe autorisées pour toutes les pièces. On s'occupera plus tard de l'élimination des coups qui correspondent à des sorties hors échiquier. Au jeu d'échecs, on peut considérer que douze directions sont possibles : les huit précédentes, plus huit « directions spéciales » par le cavalier. Une direction seulement est autorisée pour les pions, les huit sont autorisées pour la dame. Chaque direction est caractérisée par une paire (dx, dy) donnant sa pente. Ainsi, avec la numérotation précédente de ces directions :

	1	2	3	4	5	6	7	8
dx	1	1	0	-1	-1	-1	0	1
dy	0	1	1	1	0	1	-1	-1

La pièce p peut se déplacer dans chacune des « directions autorisées (p) » d'un nombre de pas au plus égal à une quantité $n_{pasmax}(p)$, qui dépend de sa nature : $n_{pasmax} = 1$ pour le roi et 8 pour la dame aux échecs ; $n_{pasmax} = 1$ pour toutes les pièces au tic-tac-toe. Si le coup associé à ce déplacement n'est pas situé hors échiquier et s'il aboutit à une case non occupée, ou encore à une case occupée par une pièce ennemie, avec prise de celle-ci autorisée par les règles, alors le coup est légal et on l'enregistre.

En outre, aux échecs, le roque, la prise en passant, l'interdiction de se mettre soi-même en échec, la promotion des pions conduisent à quelques détails techniques supplémentaires, mais le schéma de principe de la génération des coups est toujours le suivant :

```

REPETER POUR TOUTE case  $(i, j)$  de l'échiquier.
| SI occupation  $(i, j) = ami$ 
|   ALORS  $p \leftarrow pice(i, j)$ ; nature de la pièce amie en  $(i, j)$ 
|   REPETER POUR TOUTE direction  $\in [1, ndirections]$ 
|     SI direction  $\in$  directions autorisées  $(p)$ 
|       ALORS  $x \leftarrow i; y \leftarrow j$ 
|         REPETER POUR TOUT pas  $\in [1, n_{pasmax}(p)]$ 
|            $x \leftarrow x + dx(direction)$ 
|            $y \leftarrow y + dy(direction)$ 
|           SI  $(x, y) \in$  échiquier ET occupation  $(x, y) \neq ami$ 
|             ALORS enregistrer-le-coup :  $[(i, j) \rightarrow (x, y)]$ 
|             FSI
|           FR sur pas
|         FSI
|       FR sur direction
|     FSI
|   FR sur case

```

La façon la plus simple de couper, à partir de ce schéma, l'arborescence potentielle de recherche est de se contenter d'une étude à la profondeur un :

- 1) Engendrer tous les coups légaux ;
- 2) Les évaluer ;
- 3) Jouer le coup de meilleure évaluation.

En prenant en compte les fins de parties par gain ou perte immédiats, une façon possible de jouer est alors :

```

SI il existe un coup gagnant
ALORS le jouer ; FIN.
SINON considérer les autres coups amis légaux ;
soit  $c$  le coup suivant
| SI il existe un coup ennemi gagnant après  $c$ .
|   ALORS éliminer le coup  $c$ 
|   SINON compter le nombre de ripostes adverses
|     après le coup  $c$ 
|     (réappel de la procédure génération)
|   FSI
|   Jouer le coup pour lequel l'adversaire
|     possède le plus petit nombre de ripostes
FSI

```

Une stratégie « au premier niveau »

Supposons que nous ignorions l'étude résumée par la figure 6.4, qui donne au tic-tac-toe le meilleur coup par arborescence exhaustive, et choisissons d'appliquer la stratégie au premier niveau ci-dessus. Si la situation est :

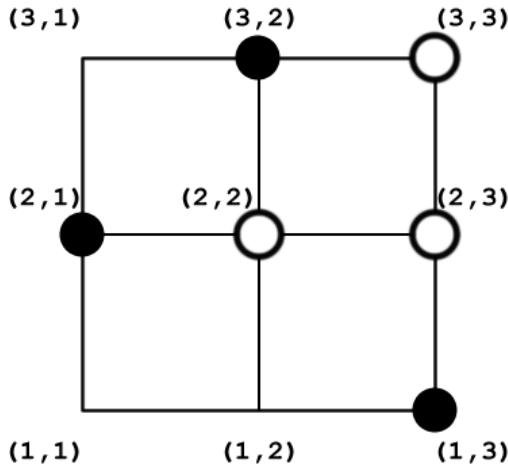


FIG. 6.4 – Trait aux noirs

L'algorithme de génération des coups légaux permet de construire la liste des quatre coups légaux :

$$\begin{aligned} a &: (3, 2) \rightarrow (3, 1) \\ b &: (2, 1) \rightarrow (3, 1) \\ c &: (2, 1) \rightarrow (1, 1) \\ d &: (1, 3) \rightarrow (1, 2) \end{aligned}$$

S'il n'existe pas ici de coup gagnant, en revanche il existe un coup, le coup d , qui provoque notre perte immédiate : nous l'éliminons donc. Parmi les trois coups qui restent l'un, le coup a , laisse quatre ripostes à l'adversaire ; les deux autres sont à égalité en laissant trois ripostes chacun. Ils sont en fait tous deux gagnants, comme on le voit en continuant à appliquer la même stratégie. L'examen d'autres positions de tic-tac-toe, montre que cette stratégie est tout à fait jouable. Mais surtout, deux points remarquables apparaissent.

Premier point : La stratégie « au premier niveau » décrite précédemment est, dans son principe, applicable à un niveau quelconque. Chaque fois qu'il est impossible de développer toute l'arborescence des coups, elle permet de comparer, à la profondeur choisie, les coups entre eux par examen d'une valeur numérique associée à la situation d'arrivée. Cette valeur numérique peut être par exemple le nombre de ripostes ennemis. Elle remplace alors la valeur logique GAIN ou PERTE qui ne pourrait venir qu'après énumération complète des positions.

Deuxième point : La stratégie au premier niveau est indépendante du jeu. Elle donne de bons résultats au tic-tac-toe et évite la mémorisation de l'arborescence exhaustive. Elle est convenable aux échecs en début de partie et dans certaines phases de jeu. Elle est équivalente à un jeu au hasard au morpion où toutes les cases libres sont équivalentes ! Mais peu importe, il est clair qu'elle peut être raffinée et spécialisée. Elle permet, dans son principe, de jouer à tout jeu à deux joueurs.

L'arborescence de la figure 6.5 résume la démonstration par énumération exhaustive : si le premier joueur place sa première pierre au centre, sa victoire est assurée. quelles que soient les ripostes adverses successives. Etant données les symétries du terrain, une fois la première pierre noire placée, seules les places a et b sont significativement différentes pour la première pierre blanche : a est équivalente à c, g et i ; b est équivalente à d, f et h . Cette remarque a été utilisée pour diviser, a priori, par quatre, la largeur

de l’arborescence. Ensuite, pour gagner du temps et de la place, nous avons seulement indiqué, pour les noirs, le meilleur coup à chaque niveau, c'est-à-dire celui qui mène sûrement à la victoire. Ainsi, l’arborescence présentée est le résultat que fournit l'énumération complète qui est facile à engendrer. Remarquons par ailleurs qu'en dehors du premier niveau, tous les coups blancs sont forcés jusqu'au huitième, où le coup blanc n'a plus d'importance ; le gain noir est assuré.

Dans tous les cas où, par contre, aucune fonction d'évaluation parfaite ne peut être construite, ou bien si le nombre de coups possibles est trop élevé pour permettre d'engendrer toute l’arborescence, **il faut limiter celle-ci**. C'est le cas du jeu d'échecs, du jeu de dames, du jacquet, du bridge, du poker, du kalah, de l'awélé, d'Othello, du morpion, du go moku, du go. Il existe deux façons de limiter la croissance de l’arborescence de tous les coups légaux : on peut la limiter **en largeur**, on peut la limiter **en profondeur**. La limitation en largeur correspond au rejet de certains coups jugés non pertinents par rapport à la position. Ainsi, aux échecs, si la dame est en prise, un coup de pion à l'autre bout de l'échiquier sera en général d'intérêt nul. Le problème est justement que, de temps en temps, un tel coup peut permettre un peu plus tard soit un échec à la découverte, soit une fourchette. Ces coups, à leur tour, pourront autoriser ou bien un mat, ou bien un coup qui récupère la dame laissée en prise, ou encore la promotion du pion poussé. On retombe donc sur le problème qu'on ne sait pas résoudre, à savoir celui de l'évaluation a priori d'une position : évaluation qui pourrait justement décider de la pertinence d'un coup... Ainsi, une telle approche de réduction en largeur éliminerait le plus souvent tous les coups de sacrifice, c'est-à-dire ceux qui donnent temporairement l'avantage à l'adversaire pour, après quelques coups, fournir une situation gagnante.

Ici les programmes actuels, dans leur quasi-totalité, se distinguent de la démarche humaine et n'opèrent absolument aucune réduction en largeur. L'homme, lui, possède une meilleure vision globale de sa situation, a en tête sa culture échiquierne, et la stratégie propre qu'il a choisi de suivre dans la partie, ce qui lui permet indubitablement de juger mieux, sinon parfaitement, la situation. Un tel jugement n'est pas impossible en machine (Cf. chapitre 6 6.5) mais la méthode est nécessairement plus complexe que l'approche brutale que nous avons choisi de décrire tout d'abord ici. Mais si la largeur ne peut être limitée, vu la croissance exponentielle inévitable et les temps de calculs rédhibitoires, **c'est sur la profondeur qu'il faut gagner**.

Une première idée consiste à reconnaître une bonne fois pour toutes qu'on ne peut aller bien loin en profondeur et que la profondeur moyenne d'étude sera tout simplement imposée, selon le calcul simple fait précédemment et à partir du temps de réponse que l'on juge raisonnable, par la puissance de l'ordinateur dont on dispose. En tout état de cause, cette profondeur sera de l'ordre de cinq et, en tout cas, inférieure à dix demi-coups (mouvements blancs ou noirs).

Une seconde idée est que cette profondeur n'a nul besoin d'être identique pour toutes les branches étudiées. En particulier, il serait tout à fait inopportun d'interrompre les recherches au beau milieu d'une série d'échanges : plus que d'autres encore, de telles positions instables sont spécialement difficiles à évaluer. Dans un même ordre d'idées, les programmes envisagent également, avant de passer à l'évaluation, l'effet de certains échecs au roi adverse, puis les nouvelles prises auxquels ceux-ci peuvent donner naissance. Ainsi la recherche est arrêtée en fin de compte lorsque, pour chaque côté, la position est jugée tranquille. La génération des coups légaux prend alors fin et la situation est évaluée.

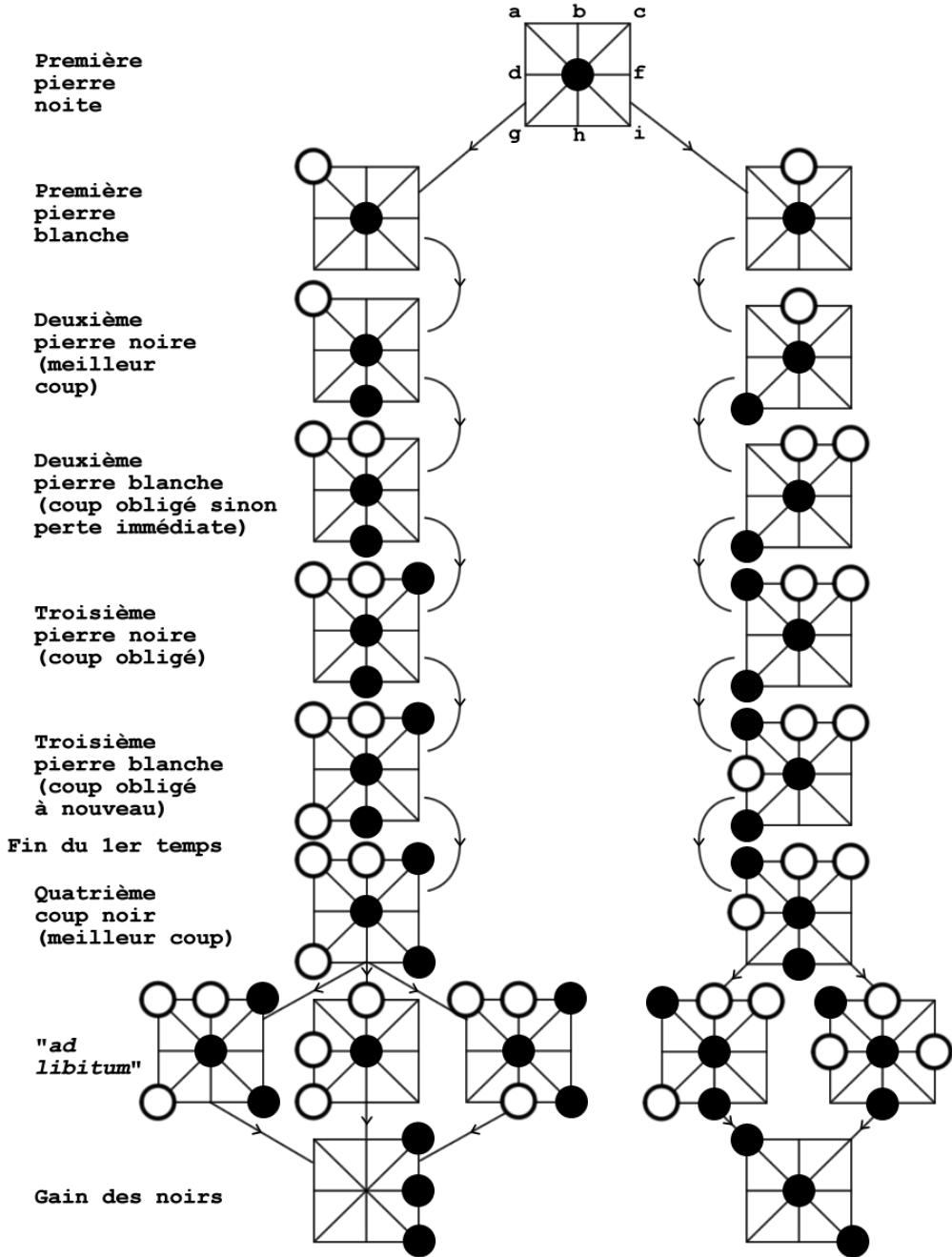


FIG. 6.5 – Gain du premier joueur au tic-ta-toe

Il est remarquable que toutes ces idées aient été pour la première fois utilisées par le mathématicien anglais Alan Turing, qui dès 1951 les avait mises en œuvre très concrètement en simulant à la main le comportement d'une telle stratégie.

6.2 L'évaluation des positions

Il s'agit maintenant de faire correspondre à toute situation S une valeur numérique $F(S)$ d'autant plus grande et négative que l'adversaire domine. Bien sûr, il n'existe aucune théorie aux échecs donnant cette fonction F exactement. Tout au plus connaît-on, pour certains débuts répertoriés, le signe du résultat. Pour tous les autres cas, tout est à inventer. Des lors, l'évaluation de certaines positions pourra malencontreusement être jugée négative alors qu'on a l'avantage. C'est là une raison fondamentale pour laquelle un programme de ce type ne peut jouer parfaitement. Avec cette approche, à chaque fois que le programme joue mal il faut trouver la raison dans la fonction F et changer l'expression de celle-ci de façon à corriger l'erreur...

La partie la plus importante dans l'évaluation est toujours le matériel : le premier terme de F est proportionnel à la somme algébrique des valeurs des pièces de part et d'autre, appelée balance B . Les valeurs retenues des pièces aux échecs sont habituellement, comme pour les joueurs humains : pion 1, cavalier et fou 3, tour 5 et dame 10. Puis d'autres termes, concernant chacun une caractéristique différente, viennent s'ajouter à F pour donner une forme polynômiale du type :

$$F(S) = a.B + b.R + c.M + d.C + e.P + f.A$$

Si le camp ami a un fou d'avance (+3) pour un pion de retard (-1) le premier terme vaut ainsi $2.a$. La valeur du coefficient a , comme celles de b , c , d , e et f est laissée à « l'astuce » de chaque programmeur... Les termes suivants sont grossièrement d'importances décroissantes : la valeur R tient compte des sûretés relatives des deux rois : R est d'autant plus grand que le roi ami est bien protégé derrière son roque. Le coefficient b doit varier en cours de partie pour aller jusqu'à s'annuler lorsque les pièces majeures ont disparu de l'échiquier, les rois devant alors soutenir leurs pions ou encore aider l'attaque en protégeant leurs propres pièces.

Le terme M mesure la mobilité des pièces : Il récapitule le nombre de cases où chacune des pièces peut se déplacer ; on pondère ce nombre par un facteur dépendant de la nature de la pièce.

Le terme suivant, C , traduit le contrôle du centre, c'est-à-dire des cases $d4$ et $d5$, $e4$ et $e5$. Le coefficient d , qui lui est associé, n'a une valeur élevée qu'en début et en milieu de partie.

La caractéristique évaluée par P concerne la structure des pions : les pions amis protégés, avancés ou passés, interviennent positivement pour le camp ami, tout comme les pions ennemis doublés, isolés ou non protégés.

Enfin (mais la liste donnée n'est pas trop limitative, quelques programmes prennent en compte d'autres caractéristiques) le dernier terme, en A , fait le bilan des possibilités d'attaques en tenant compte des pièces avancées au-delà de la quatrième ligne et protégées, des tours sur des colonnes ouvertes, des tours doublées, des possibilités d'échange contre des pièces adverses de grande mobilité, des attaques sur les cases voisines du roi ennemi, des possibilités d'échecs et enfin des clouages.

La mise au point des coefficients de pondération de a à f est une affaire de patience et rien n'interdit de faire varier leurs valeurs en cours de partie en fonction du nombre de coups joués depuis le début et/ou en fonction des pièces restant sur l'échiquier. Il est alors, on le voit, dans la nature même de la fonction d'évaluation de permettre des sacrifices positionnée, c'est-à-dire d'amener des positions où la caractéristique B a une valeur négative (une pièce en moins que l'adversaire) mais dans laquelle la somme pondérée des autres termes compense avec bénéfice cette différence défavorable de matériel.

Les valeurs des coefficients a à f peuvent également être modifiées directement par le

programme lui-même en fonction des résultats (voir chapitre 9).

Si la fonction F était parfaite, il suffirait d'engendrer tous les coups amis légaux, d'évaluer les situations auxquelles ils conduisent, et de jouer celui qui correspond à la meilleure d'entre elles. En pratique, on est loin de savoir construire d'aussi bonnes fonctions. Comme, par ailleurs, on ne peut développer l'arbre assez loin pour atteindre des positions où l'avantage de l'un ou de l'autre est indiscutable, il faut trouver un **compromis** entre ces deux extrêmes, compromis entre la profondeur (moyenne) de la recherche et la qualité de la fonction d'évaluation. En effet, augmenter chacune de ces grandeurs contribue pareillement à la croissance du temps de calcul !

En tout état de cause, un autre problème doit encore être traité, qui consiste à savoir déduire à partir des positions terminales atteintes à profondeur maximale, le coup à jouer effectivement au premier niveau. La méthode qui fait l'unanimité depuis que les machines ont été programmées pour jouer, et quel que soit le jeu, est connue sous le nom de *minmax*.

6.3 Le *minmax* et le choix du coup à jouer

Remarquons tout d'abord que, précisément parce que notre fonction d'évaluation ne saurait être parfaite dès la position initiale, il ne servirait à rien d'évaluer les positions intermédiaires à mesure qu'elles sont rencontrées en cours de descente de l'arborescence de recherche, tant que la profondeur maximale choisie, soit n , n'est pas encore atteinte : ces positions risquent fort, notamment, d'être instables et d'évaluation incommode. Une fois parvenu à la profondeur n , il convient de prendre encore en compte toutes les captures dans l'ordre convenable de valeur des pièces et les échecs. Ceci s'appelle le mécanisme de *quiescence* (de « quiet » = tranquille). Ainsi dans la figure 6.6, les branches terminales de l'arbre ne sont pas toutes de la *même* longueur : celle-ci dépend du nombre de coups qu'il a fallu étudier pour atteindre une position tranquille. C'est alors seulement, en parcourant en arrière l'arborescence des coups, que nous attribuerons des valeurs aux sommets qui précèdent chronologiquement les situations associées aux feuilles. Mais, comment procéder ? Oscar Morgenstern et John Von Neumann avaient, en 1945, proposé une procédure valable pour tout jeu de stratégie.

L'hypothèse fondamentale supplémentaire ici est que l'adversaire s'en remet à la même fonction d'évaluation que nous-mêmes.

Dès lors nous devons **maximiser** la valeur de la fonction d'évaluation de la situation courante, alors qu'en revanche il convient, pour notre adversaire, de **minimiser** celle-ci. Ainsi, à tout niveau où c'est à lui de jouer, dans le pire des cas, l'adversaire retiendra le coup qui mène à la situation d'évaluation la plus faible. Tout se passe donc comme si l'évaluation à ce niveau, avant que l'adversaire n'ait joué, était le minimum des évaluations qui lui succèdent immédiatement. Quand le niveau, au contraire, correspond à un trait ami, c'est le maximum des évaluations un niveau plus bas qui doit être retenu, d'où le nom de *minmax* donné à la procédure de Morgenstern et Von Neumann. De proche en proche, tous les noeuds intermédiaires reçoivent ainsi une évaluation. Quand tous les successeurs d'un noeud sont connus, la valeur maximum ou minimum d'entre eux, suivant la parité du niveau, lui est finalement attribuée. Remontant ainsi les évaluations jusqu'à la racine de l'arborescence, c'est-à-dire jusqu'à la position initiale, on en déduit au bout du compte le meilleur coup ami, qui correspond à la position au premier niveau dont l'évaluation remontée est la plus forte. Il est important de noter que cette évaluation n'a, a priori, absolument rien à voir avec celle obtenue directement

en appliquant F à la position initiale sans tenir compte des coups ultérieurs : on a bien une évaluation corrigée.

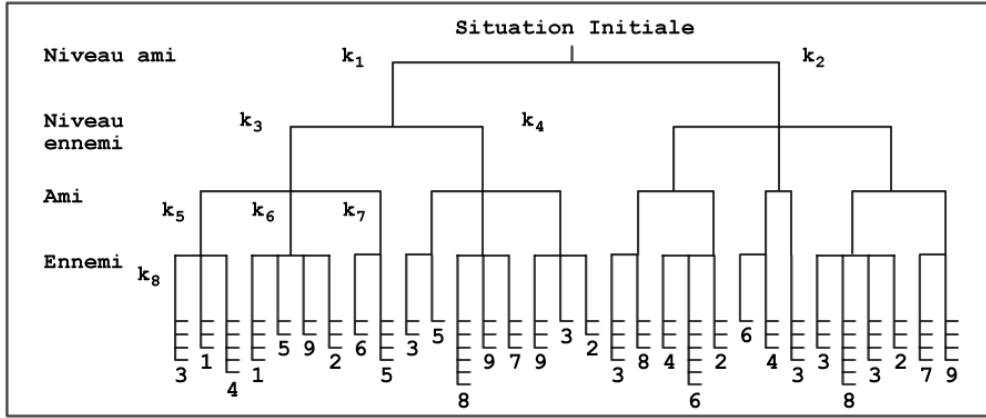


FIG. 6.6 – Une arborescence des coups légaux avec évaluation à profondeur variable des positions stables

La figure 6.7 donne un exemple d'étude et de résultat fourni par le minmax. Deux coups légaux seulement, soit k_1 et k_2 , se présentent sur la position initiale où le trait est au camp ami. A partir de la position p_1 , atteinte après le coup k_1 , on engendre tous les coups légaux possibles, soit k_3 et k_4 . Etudiant d'abord la variante associée à k_3 , on engendre à nouveau tous les coups à partir de la situation correspondante. Trois coups, k_5 , k_6 et k_7 sont ainsi produits. Le coup k_6 mène à une position où trois autres coups, k_8 , k_9 et k_{10} sont, à leur tour, engendrés.

Supposons que, pour notre programme, 4 soit la profondeur maximale. Chaque situation après k_8 , puis k_9 , puis k_{10} , est donc étudiée, jusqu'à élimination des prises et des principaux échecs au roi, et évaluée. Les évaluations données par la fonction F sont portées dans la figure en bas de l'arbre. Si le coup ami a le trait sur la position initiale, c'est le camp ennemi qui avait le choix du coup au dernier niveau développé : des trois évaluations associées (3, 1 et 4), il retiendra donc (du moins s'il joue avec la même évaluation que nous) la plus faible, c'est-à-dire 1.

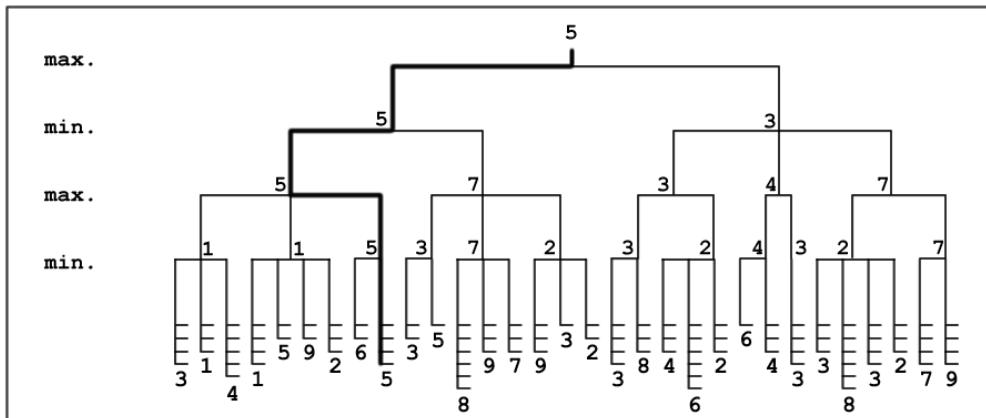


FIG. 6.7 – Remontée des évaluations par *minmax* pour l'arborescence précédente

Ecrivons l'algorithme du *minmax* proprement dit :

MINMAX : {Descendre jusqu'à la profondeur maximale, alors évaluer la position ; comparer à *min*, ou *max*, suivant la profondeur ; passer au corps suivant au même niveau s'il en existe, sinon remonter d'un niveau.}

Nous avons besoin de plusieurs procédures de service :

- *Engendrer-coups* (*échiquier, loueur*) : engendre les coups légaux pour l'échiquier en cours et le joueur qui a le trait ; range ces coups dans une pile : *pilecoups* (*échiquier, camp*).
- *Jouer* (*échiquier, coup*) : met à jour l'échiquier lorsque le coup joué est donné.
- *Déjouer* (*échiquier, coup*) : c'est la fonction inverse de la précédente : elle rétablit la situation avant le coup.
- *Valuation* (*échiquier, prof*) : c'est la valeur numérique associée à la position.

Pour simplifier la description de l'algorithme, on multiplie systématiquement les valuations impaires par -1 : ainsi il suffit de prendre le **maximum** des évaluations quel que soit le niveau.

En outre : *prof* désignera la profondeur en cours ; *profmax* désignera la profondeur maximale de recherche ; *valuation* (*échiquier*) calcule la fonction d'évaluation de la position ; *eval* (*prof*) est la meilleure valeur connue à ce niveau ; *E* (*prof*) contient la liste des coups légaux en cours d'étude (cf. table 6.1).

La valeur 1 devient donc celle de la situation obtenue si nous prenons la décision de jouer le coup k_5 au troisième niveau. De la même façon, nous ne pouvons compter, après le coup k_6 , sur un résultat de valeur 1 puisque l'adversaire a, encore une fois, une riposte qui lui assure une position où il peut limiter nos ambitions à un score d'une unité. Le coup k_7 enfin, est à ce niveau légèrement supérieur, sous les hypothèses correspondant au jeu des coups des niveaux un et deux. En effet, au mieux, notre adversaire atteint une position de valeur 5. Les trois positions issues de la position amenée par k_1 puis k_3 ont maintenant reçu une valeur. Or, dans cette position, nous avons le trait : nous ferons donc en sorte, si cette position se présente dans la suite de la partie réelle, de jouer le coup qui maximise la valeur atteinte : entre 1, -1 et 5, nous retiendrons 5, qui correspond au coup k_7 . Cette valeur devient donc l'évaluation, remontée par *minmax*, associée à la position après k_1 suivi de k_3 .

La suite et la fin de la procédure s'effectuent pareillement et la figure 6.7 donne les résultats. Sur cette figure, on trouve, à côté de chaque position intermédiaire, l'évaluation calculée par *minmax*. En particulier, l'évaluation de la position initiale vaut 1.

Remarquons que, dans cette procédure, tout comme pour le problème des huit dames, il convient, lors de la descente dans l'arborescence, de jouer les coups et les « déjouer » lors de la remontée. Mais il suffit pour cela de tenir à jour l'échiquier et de retenir en mémoire les coups qui, à chaque niveau, ont déjà été examinés. Au total, si le temps d'exécution peut être important, car il croît très vite, nous l'avons vu, avec la profondeur de recherche, la place nécessaire en mémoire reste toujours quant à elle relativement faible : toute la partie de l'arborescence et tous les échiquiers qui ne correspondent pas à la seule suite des coups présentement examinés, n'ont pas à être mémorisés : seule la meilleure valeur numérique à chaque niveau est utile. C'est pourquoi même des micro-ordinateurs bon marché mènent cette tâche à bien.

Il existe une procédure, équivalente théoriquement au *minmax*, puisqu'elle fournit toujours rigoureusement le même résultat, mais nettement plus rapide car elle permet d'éliminer définitivement, sans avoir à les examiner, des parties entières de l'arborescence. Cette procédure utilisait deux variables nommées α et β dans la publication où elle fut pour la première fois décrite en 1961, d'après une idée de John Mac Carthy : ces

<pre> MINMAX : DONNEES : (échiquier, camp) RESULTAT : (valeur de la position = <i>minmax</i>) <i>prof</i> \leftarrow 1 ; <i>camp</i> \leftarrow 1 ; <i>E</i>(1) \leftarrow pilecoups(<i>chiquier</i>, <i>camp</i>) <i>eval</i>(1) \leftarrow $-\infty$ REPETER TANT QUE <i>prof</i> \geq 1 REPETER TANT QUE <i>E</i>(<i>prof</i>) \neq vide <i>coup</i> \leftarrow sommet - pile(<i>E</i>(<i>prof</i>)) <i>chiquier</i> \leftarrow jouer(<i>chiquier</i>, <i>coup</i>) SI <i>prof</i> = <i>profmax</i> ALORS <i>prof</i> \leftarrow <i>prof</i> + 1 <i>camp</i> \leftarrow <i>camp</i> - 1 <i>E</i>(<i>prof</i>) \leftarrow pilecoups(<i>chiquier</i>, <i>coup</i>) <i>eval</i>(<i>prof</i>) \leftarrow $-\infty$ SINON <i>profmax</i> : évaluer la position et comparer <i>eval</i>(<i>prof</i>) \leftarrow MAX[<i>eval</i>(<i>prof</i>), <i>camp</i> * valuation(<i>chi</i>)] (déjouer le dernier coup à <i>profmax</i>) <i>coup</i> \leftarrow dpile(<i>E</i>(<i>prof</i>)) <i>chiquier</i> \leftarrow djouer(<i>chiquier</i>, <i>coup</i>) (rester à cette profondeur tant qu'il reste des coups) FSI FR sur <i>E</i> (Retour arrière) SI <i>prof</i> = 1 ALORS <i>minmax</i> \leftarrow <i>eval</i>(1) ; fin FSI. <i>camp</i> \leftarrow $-camp$ <i>prof</i> \leftarrow <i>prof</i> - 1 (Déjouer le dernier coup à cette profondeur) <i>coup</i> \leftarrow dpile(<i>E</i>(<i>prof</i>)) <i>chiquier</i> \leftarrow djouer(<i>chiquier</i>, <i>coup</i>) (Remonter l'évaluation par minmax) <i>eval</i>(<i>prof</i>) \leftarrow MAX[<i>eval</i>(<i>prof</i>) - <i>eval</i>(<i>prof</i> + 1)] FR sur <i>prof</i> FIN minmax </pre>

TAB. 6.1 – Algorithme *minmax*

variables sont à l'origine de son nom curieux. Mais même ainsi améliorée, la procédure reste entachée d'un défaut essentiel ; une partie est jouée coup à coup sans aucun plan d'ensemble. La force de l'homme réside précisément dans sa capacité de concevoir, de développer, d'adapter des stratégies.

Un autre type de procédure, plus proche du raisonnement humain, est ainsi programmable. Les chercheurs en Intelligence artificielle s'efforcent aujourd'hui d'écrire des programmes de cette famille ; dans le cas du jeu d'échecs, c'est toute la connaissance des joueurs accumulée depuis des siècles qui doit être donnée ; elle s'exprime en termes de clouage, d'attaque double, de colonne ouverte, de faiblesse du roque, etc. Tous ces termes, qui sont ceux du langage habituel d'un joueur, doivent être compris et manipulés par le programme.

De tels programmes, basés sur la connaissance et non plus sur la force brute, commencent à exister dans plusieurs autres domaines ; chimie, médecine, mathématiques (Cf. chapitre 7 et chapitre 6.6.8).

6.4 La procédure Alpha-Bêta

L'idée fondamentale est ici, tout comme dans les algorithmes pour la coloration optimale et le voyageur de commerce, d'utiliser les meilleures valeurs déjà trouvées dans les branches complètement étudiées, pour les comparer aux meilleures valeurs espérées dans les branches qui restent. Sous certaines conditions, qui vont être maintenant étudiées, on prouve alors que certains développements sont absolument superflus. Le minmax était une énumération explicite de l'espace des coups. La procédure alpha-bêta n'en fera qu'une énumération implicite, à l'aide de *coupes*.

La figure 6.8 présente une première situation où une telle coupe est possible. Le noeud S correspond à une position, où le trait est au camp ami, qui dispose de plusieurs coups possibles, dont l'un mène à une position A et l'autre à une position Y . Nous supposons que A a déjà été complètement analysée et que sa valeur $\langle \alpha \rangle$ est connue. La position Y est alors examinée : le premier coup étudié correspond à la situation Z ; l'évaluation de Z , calculée par *minmax*, est trouvée égale à z .

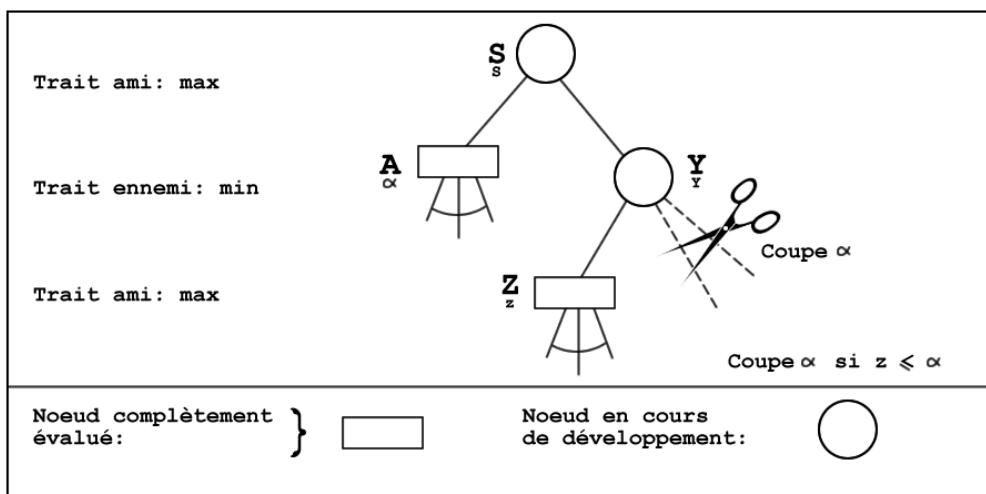


FIG. 6.8 – Coupe superficielle α

Supposons alors que la valeur z soit telle que $z \leq \alpha$. Soient, de plus, respectivement s et y les valeurs, pour le moment inconnues, des noeuds S et Y . Dans cette hypothèse, puisqu'au noeud Y l'adversaire choisira le coup menant à la position de valeur minimale, on aura en tout cas : $y \leq z$.

La situation est alors : $y \leq z \leq \alpha$, si bien que s , qui est lui-même supérieur ou égal à α et à y , vérifie la relation : $y \leq z \leq \alpha \leq s$. Ces inégalités montrent que la valeur inconnue y du noeud Y n'interviendra pas : l'étude de tous les successeurs de Y autres que Z est alors inutile. La conclusion est déjà acquise localement pour cette partie de l'arborescence et le développement de cette dernière peut ainsi, au sommet Y , être coupé après l'étude du sommet Z (*coup de type α*).

Le raisonnement qui vient d'être fait s'étend trivialement au cas où le premier niveau est associé au trait de l'adversaire, si, cette fois, l'évaluation ω , à la profondeur 2, vérifie $\omega \geq \beta$ (*coup de type β*). Mais, de plus, il se généralise également à des noeuds A et Z qui se trouvent à des profondeurs très différentes dans l'arborescence, du moment qu'ils sont à des niveaux de même parité. La figure 6.9 présente le cas général.

Soient donc, de nouveau, A et U deux positions possibles après le coup ami joué en S . A a déjà été développé et l'évaluation remontée est α . Le début de l'étude de U

conduit, après un nombre impair, ici 3, de coups intermédiaires, à la position 2 dont l'évaluation est z . Supposons à nouveau que z soit trouvée inférieure à α . Au niveau immédiatement supérieur on trouvera, pour le noeud Y , une évaluation y qui vérifiera nécessairement, le trait étant ici à l'adversaire, la relation : $y \leq z$. Nous voulons montrer qu'ici encore, l'étude des positions succédant au sommet Y à la position Z ne peut modifier le résultat final, c'est-à-dire l'évaluation de S .

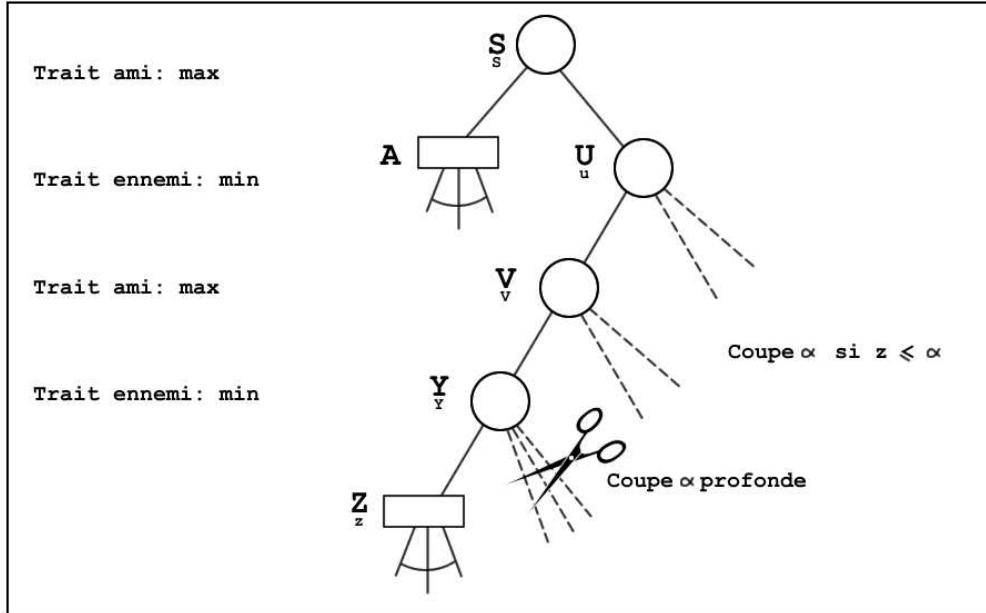
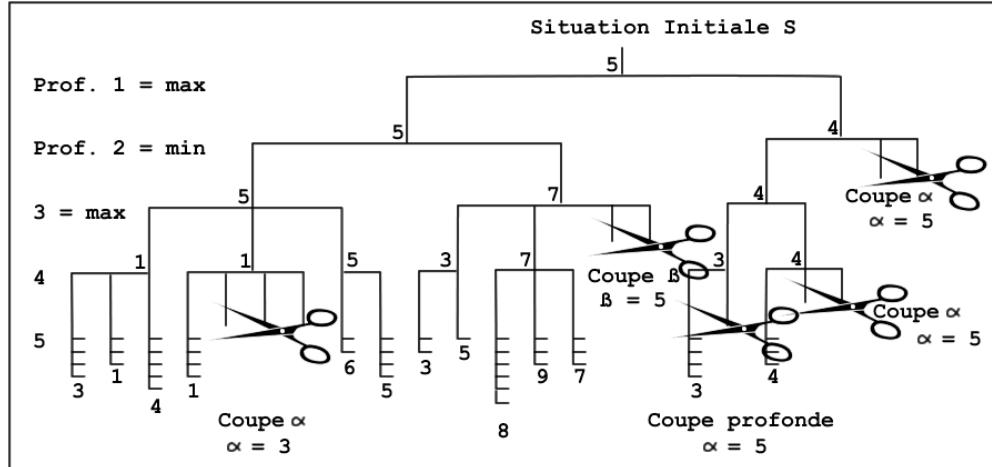


FIG. 6.9 – Coupe profonde α

À un noeud V , deux éventualités peuvent se présenter : ou bien l'évaluation maximale v serait finalement strictement supérieure à y , ou bien le maximum en V serait précisément obtenu pour cette valeur y . Dans le premier cas, puisqu'on a déjà $y \leq z$ et que l'analyse des autres coups possibles en Y ne saurait que faire diminuer encore y , il est clair que l'analyse de ces autres coups peut être omise sans changer l'évaluation v . Dans le second cas, on a : $y = v$ et, dès lors, il vient avec les inégalités précédentes : $v = y \leq z \leq \alpha$. Si bien que dans ce cas, le noeud V recevra une évaluation inférieure à celle du noeud A et on est donc ramené au schéma de la figure 6.8. Finalement, dans les deux cas, la valeur s du noeud S est indépendante des autres successeurs de Y , qui peuvent donc encore être ignorés (coupe profonde α).

Ce dernier raisonnement se généralise enfin par récurrence sur la profondeur du noeud Z , quand elle reste de même parité que celle de la racine S . Quand S correspond à un trait ennemi, c'est bien sûr la valeur minimale β des fils déjà étudiés qui sert de borne. Les *coupes de type β* ont lieu chaque fois qu'un noeud, obtenu après un trait adverse, reçoit une évaluation, soit ω , cette fois supérieure à la borne β . On voit donc qu'à un instant donné, dans le parcours de l'arbre, on compare un noeud de profondeur paire à la meilleure évaluation paire connue, soit α , et un noeud pair à la meilleure évaluation impaire, soit β . Ces deux valeurs, α et β initialement égales respectivement à plus et moins l'infini, suffisent ainsi pour remonter à la racine l'évaluation finale et pour choisir le coup à jouer.

FIG. 6.10 – Procédure α - β sur l'arborescence de la figure 6.6

En résumé :

Trait	Meilleure évaluation actuelle	Noeud à la profondeur n	Evaluation de ce noeud	avec		Résultat
ami	α	ami	z	$z = \alpha$	\Rightarrow	coupe α
ennemi	β	ennemi	w	$w = \beta$	\Rightarrow	coupe β

Puisque ces coups se produisent à des profondeurs quelconques dans l'arborescence, sur des noeuds qui doivent simplement être de même parité, il n'est besoin que de retenir la meilleure borne connue pour les niveaux pairs (α), et la meilleure borne connue pour les niveaux impairs (β), à partir de la racine, unique sommet de niveau zéro.

Aucune coupe n'aura lieu tant que deux noeuds terminaux au moins n'auront pas été mesurés par la fonction d'évaluation F . Lors de l'exécution de la procédure, les valeurs de α pour chaque niveau ne cessent de croître, celles de β de décroître. La fin est caractérisée par leur égalité.

La procédure α - β est un algorithme exact, plus rapide que le *minmax*, pour un même résultat. Il donne de bons résultats lorsque la fonction d'évaluation est bonne et que la profondeur est assez grande. Ces deux qualités ne peuvent être obtenues qu'à l'aide d'un effort de calcul important. Le succès de l' α - β est cependant évident pour les mats en n coups (n actuellement inférieur à 6) puisqu'alors la fonction d'évaluation est exacte et que la profondeur maximale peut être atteinte en un temps acceptable. Dans le cas général, la qualité de jeux (cf. micro-ordinateurs actuels) va de moyenne à bonne. Soulignons enfin un phénomène amusant : la procédure α - β effectue d'autant plus de coups - et est d'autant plus rapide - que les sommets sont mieux classés pour chaque coup. Le classement idéal, bien entendu inconnu, rangerait à chaque niveau les coups par valeur décroissante. Pour approcher ce classement certains programmes, avant de lancer l' α - β , disons à la profondeur 6, le lancent à la profondeur 2 et reclassent alors les coups en fonctions du résultat. Puis l' α - β est de nouveau relancé à la profondeur 3 par exemple. Le nouveau classement est mis en mémoire et c'est seulement à partir de lui que la procédure est lancée à la profondeur 6. Les deux premiers essais qui travaillent avec un petit nombre de sommets sont très rapides, mais permettent d'approcher le classement idéal pour le niveau 6, si bien qu'au total, pour la même solution, le temps de calcul est plus court qu'en lançant la procédure directement au niveau maximal.

6.5 Les défauts fondamentaux liés au développement systématique de l'arbre des coups

Les défauts liés à l'énumération systématique sont à la fois nombreux et graves. On peut même s'étonner de la qualité du jeu des programmes correspondants malgré l'énorme handicap que ces défauts impliquent. Les deux défauts principaux sont présentés ci-après.

6.5.1 Premier défaut : le manque total de stratégie

Par l'approche même, chaque position est nouvelle : ces programmes **ne jouent pas une partie**, mais une succession de positions entièrement indépendantes les unes des autres. Il est clair que l' α - β ne permet pas de propager un plan ou une idée stratégique quelconque. Tout le jugement est contenu dans la fonction d'évaluation ! Seuls quelques termes, simples à calculer, évaluent numériquement la qualité positionnelle des pièces amies sur l'échiquier : développement, lignes d'attaque, pion soutenus, pions passés. Mais ces termes ne peuvent en aucun cas traduire une vision à long terme de la partie, ni permettre de raisonner en fonction de telle ou telle finale a priori avantageuse. Les exemples les plus simples qui montrent bien ce défaut se rencontrent souvent en fin de partie. Considérons la position à trois pions contre deux de la figure 6.11 :

Ici tout amateur voit que le gain des blancs est assuré. Mais ce gain nécessite en

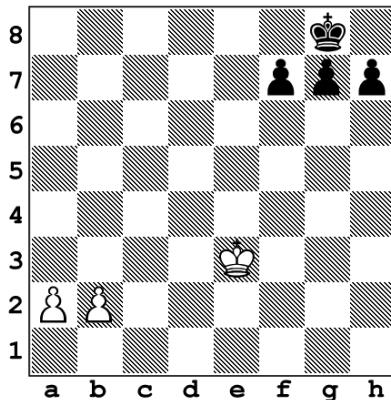


FIG. 6.11 – Finale avec trait aux blancs

réalité un plan à long terme : pousser le pion $a2$ à dame. Le coup $a2-a4$ assure ainsi aux blancs la victoire finale. Ce coup, pour un homme, est immédiat. Il entraîne, ipso facto, l'abandon du possesseur des pièces noires : le roi noir ne peut plus rentrer dans le carré du pion $a4$ (carré $a4, a8, e8, e4$). Donc $a4$ peut aller à dame tout seul, sans la protection du roi. Comme par ailleurs aucun pion noir ne peut faire de même, la partie des noirs est perdue. Les programmes du type précédent ne peuvent, quant à eux, se livrer à une telle analyse, même aussi simple, et 10 demi-coups d'avance (avec un coup forcé du roi blanc pour parer un échec intermédiaire) ne suffisent pas pour conduire $a2$ à dame ; leur conclusion est claire : avantage aux noirs par trois pions contre deux ! Ainsi, seule la force brute, la puissance gigantesque de calcul de l'ordinateur est utilisée dans les programmes de jeux de cette première famille, et aucunement ses capacités de raisonnement formel avec traitement de schémas d'échiquiers et manipulation effective de concepts adéquats. C'est une pure vision anthropomorphique

qui nous fait quelquefois croire, dans les parties d'échecs contre des programmes de ce type, que le programme a conçu une belle idée d'attaque sur telle ou telle pièce... idée qu'il abandonne deux coups plus tard, d'autant plus facilement que lui, en fait, ne l'a jamais eue !

Les programmes de cette famille sont d'autant plus forts qu'ils peuvent développer un plus grand nombre de coups à la seconde : qu'il s'agisse de calculer une racine carrée ou de jouer aux échecs, leurs rapports de performances sont semblables et la seule différence entre un microprocesseur et un gros ordinateur est la vitesse de calcul : l'un étudie aujourd'hui (1986) un millier de coups, l'autre cinq cent mille sur ordinateur CYBER, en un temps de l'ordre de la minute. Mais si l'approche est le développement systématique des coups légaux, même des performances de cette grandeur ne sauraient suffire.

6.5.2 Deuxième défaut : l'effet d'horizon

Il y a deux inconvénients majeurs au fait que l'on doive fixer a priori une profondeur maximum (de quelques demi-coups) à la recherche par énumération des coups légaux. Le premier est évident : le programme ne peut « voir » (et, a fortiori, évaluer) les effets d'un coup qui se produisent à une profondeur dépassant celle qu'il a considéré. Le second est plus subtil, il a été souligné, en particulier, par Hans Berliner (ancien champion du monde des échecs par correspondance) dans sa thèse en 1974. C'est un corollaire inévitable, mais diabolique, de l'effet précédent. On l'appelle l'effet d'horizon : il est dû au fait qu'un programme de la famille énumération-systématique aura, par nature, tendance à tout faire pour repousser au-delà de son horizon tout événement défavorable. Il ne veut pas voir une catastrophe qui arrive, il dispose d'un moyen pour cela : repousser l'échéance par des coups absurdes !...

Examinons un exemple tiré de l'excellent Ph. D de H. Berliner : (figure 6.12)

Le fou blanc en *a4* est cerné par la meute des pions noirs : il est perdu. La stratégie d'un bon joueur humain serait alors, pour les blancs, de récupérer le maximum pour cette perte imparable : il est possible de regagner un pion et d'en attaquer un autre (la situation sera alors excellente pour les blancs qui ont, dans la position donnée, un fou d'avance).

Notation au jeu d'échecs : C = cavalier, F = fou, T = tour, D = dame, R = roi, le pion n'est pas marqué ; colonnes : *a* à *h*, lignes 1 à 8 ; le signe \times indique une prise, + un échec. Ainsi : 1 - *e5* indique que le pion de la colonne *e* avance d'un pas au premier coup (la case de départ n'est pas indiquée) ; ou bien : 3 - *C×Fe7+* signifie que le cavalier prend, au 3ème coup, le fou situé sur la case *e7* en donnant échec.

Pour un programme d'énumération qui joue les blancs, il n'en va pas ainsi : il lui suffit de s'arranger pour que la capture du fou soit **repoussée au-delà de son champ de vision, de son horizon**, et tout se passera alors comme si ce fou ne devait jamais être pris. Ceci est dans la logique de la merveilleuse mécanique de l' α - β ; Il suffit de trouver des coups en assez grand nombre, avec riposte forcée, éloignant l'adversaire de la prise. Ainsi, sur la position précédente, si la profondeur est fixée à cinq demi-coups. le meilleur coup remonté par α - β sera *e5* ! Ce coup correspond au « sauvetage » suivant :

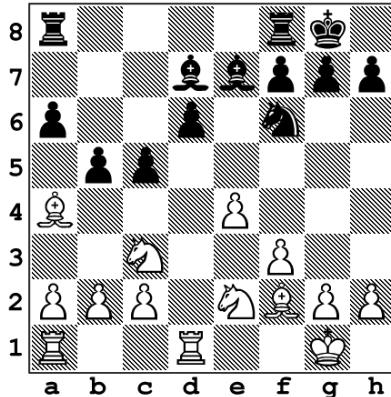


FIG. 6.12 – Effet d’horizon, H. Berliner (1974). Trait aux blancs

1-e5	$d \times e$	(sinon on gagne le <i>Cf7</i>)
2- <i>Cd5</i> et si	$C \times Cd5,$	alors 3- <i>T</i> × <i>Cd5</i> (a)
	<i>Fe7</i> joué,	alors 3- <i>Fb3</i> (b)
	<i>b</i> × <i>Fa4,</i>	alors 3- <i>C</i> × <i>Fe7+</i> (c)

Dans le cas (a), le mécanisme de quiescence donne, certes, avant évaluation de la position, $b5 \times Fa4$, mais il fournit aussi $T \times Fd7$, coup que les noirs empocheront en réalité par : 4-*Fe6*. Au total, il indique la prise d’un pion au lieu d’un fou ! Dans le cas (b), le programme joue *Fb3*, bien sûr, avant d’évaluer. Les deux cavaliers sont protégés et le fou est sauf ! En fait il est perdu à cause de 3-*c4*. Enfin, dans le cas (c), la prise noire est prématurée et les blancs gagnent définitivement la pièce. Le jeu correct au premier niveau n’apparaît que si le programme joue jusqu’à *profmax* = 7. Il lui faudrait, dans ces conditions, étudier 168 800 échiquiers différents !

Aucune parade définitive à cet effet d’horizon - dont les ravages, on le pressent, peuvent être grands - ne paraît concevable. Il naît, en fait, de l’approche elle-même. Ses conséquences sont les plus désarmantes et ce sont elles qui éloignent le plus ces programmes du comportement du joueur humain. Elles peuvent être très éprouvantes pour le programmeur qui regarde jouer son programme : lors d’une belle partie, le programme *Chess4.4* possède, ainsi, un pion passé en septième rangée. Ce pion vaut cher pour *Chess4.4*, mais en réalité, il est perdu, impossible à protéger. Pourtant, *Chess4.4* ne « veut » pas le perdre, repousse la prise... et perd son net avantage après une suite d’échanges défavorables. Enfin l’adversaire lui prend le pion... *Chess4.4* retrouve alors tous ses moyens, se remet à bien jouer, et gagne !

6.6 Les études des psychologues sur la résolution de problèmes par les hommes

Il n’est pas facile de dater avec exactitude les premiers travaux des psychologues dans le domaine de la résolution de problèmes. En France, Alfred Binet avait, dès 1894, écrit un traité : « La psychologie des grands calculateurs et des joueurs d’échecs ». Il y étudiait en particulier, en remarquable pionnier qu’il était, les importances respectives de la mémoire, de l’imagination et de l’érudition. Il s’intéressait aux représentations mentales abstraites utilisées en calcul mental et au jeu d’échecs des sujets qui voulaient bien se prêter à ses expériences. Pour cela, il les faisait calculer ou jouer aux échecs

« à l'aveugle » et leur demandait d'analyser à haute voix.

Les études de Binet seront reprises et complétées par la suite par plusieurs psychologues comme Adrian De Groot, Herbert Simon, et John Baylor tout spécialement pour les joueurs d'échecs ; Bergson, premier commentateur de Binet, note en 1902 un fait crucial : les représentations semblent, dit-il, contenir moins d'images du monde physique lui-même (pièces et échiquier) que d'indications sur les **prochaines directions à suivre**.

Otto Selz, psychologue de l'école de Wurzburg et de la Denkpsychologie, qui est assez peu connu, élaborait pourtant dès 1920 un premier modèle conceptuel des processus de pensée décrits comme chaînes linéaires d'opérations élémentaires. Selz isolait ainsi les opérations et les caractérisait, tandis qu'en même temps il décrivait précisément les processus généraux valides, lui semblait-il, pour tout le genre humain. Il n'en notait pas moins l'importance, au cours de la résolution d'un problème, des paramètres suivants : la personnalité intellectuelle du sujet, les caractéristiques particulières du problème, la détermination du sujet à résoudre le problème. Le « moteur » qui permettait à ces chaînes de se développer lui semblait être « la loi générale d'anticipation des résultats », autrement dit, le raisonnement par plans et par sous-but, avec un degré convenable d'abstraction.

Le mérite et la chance des chercheurs en intelligence artificielle consiste, aujourd'hui, à pouvoir tester effectivement par programme la pertinence de telles idées. Mais on voit à quel point des hommes comme Binet et Selz étaient des précurseurs : les premiers, sans doute. Ils ont considéré **la pensée et le raisonnement comme des phénomènes d'expérience**, au même titre que d'autres, propres à être approchés scientifiquement. Ils avaient déjà montré, au seuil du vingtième siècle, que nos facultés intellectuelles étaient d'abord liées et conditionnées par deux phénomènes bien tangibles : la mémoire et la perception.

6.6.1 La mémoire

Les plus gros ordinateurs possèdent aujourd'hui une mémoire centrale qui compte de dix millions à un milliard d'unités binaires d'informations. Mémoire centrale est un terme qui indique que ces informations sont accessibles de façon permanente à l'ordinateur : tout ce que nous avons, nous, dans le cerveau est en mémoire centrale. Ce terme s'oppose aux mémoires auxiliaires : disques, cassettes, bandes, qui jouent ni plus ni moins le rôle de nos livres dans nos bibliothèques : ces supports secondaires d'information sont moins faciles d'accès.

Le temps moyen d'accès à une information binaire en mémoire centrale dans un ordinateur est actuellement d'environ un milliardième de seconde : les ordinateurs y effectuent très vite les quelques actions élémentaires qu'ils connaissent. En revanche, il serait faux de croire qu'ils nous surpassent aussi par la taille de leur mémoire : le cerveau humain (moyen...) possède cent milliards de *neurones* : chacun d'eux est relié par des connexions (*synapses*) à mille ou dix mille - on ne sait dire actuellement - de ses petits camarades. Il existe ainsi dans un cerveau **cent mille milliards de points d'échanges d'informations** (incroyable, non ? Vous mesuriez peut-être mal votre seule vraie richesse!...) Or en outre, personne aujourd'hui ne sait évaluer la capacité de rétention d'informations d'un neurone : mais, sans aucun doute, est-elle beaucoup plus grande que l'unité binaire correspondant à la valeur d'une information de type oui/non. Peut-être un neurone contient-il mille ou un million de telles informations binaires. Peut-être la même information est-elle codée, à un moment donné,

dans plusieurs neurones à la fois. En tout état de cause, il est certain que nous avons été construits avec une mémoire gigantesque, notamment par rapport aux mémoires centrales des plus grands ordinateurs actuels. Cette capacité énorme suppose que nous soyons par ailleurs dotés de **mécanismes d'accès à ces informations élaborés et efficaces**. C'est ainsi qu'il n'est guère étonnant de constater expérimentalement que nous structurons sans cesse notre connaissance, nous construisons des points de repères multiples. nous donnons des noms à de nouvelles entités, nous codons, nous sélectionnons, enfin... nous oublions !

L'expérience montre également qu'une partie du cerveau est utilisée de manière très particulière : il s'agit de la gestion de notre *mémoire à court terme*. L'ensemble des neurones qui constituent celle-ci est en **communication directe** et continue avec les organes des sens. C'est par elle que transitent aussi nécessairement toutes les nouvelles informations avant d'être emmagasinées dans l'autre partie de notre mémoire : la *mémoire à long terme*. Dans la mémoire à court terme, les informations se détériorent très vite : après cinq secondes, elles sont détruites si elles n'ont pas été réactivées. Dans la mémoire à long terme, les informations subsistent beaucoup plus longtemps : des heures, des jours, des années, sans être réactivées autrement que par un processus purement interne et inconscient, qui est peut-être à la naissance de nos rêves... Inversement, une opération d'écriture prend plus de temps dans la mémoire à long terme : elle demande plusieurs secondes contre quelques centièmes de seconde dans la mémoire à court terme.

6.6.2 La perception

Ces quelques faits font peut-être mieux comprendre nos attitudes et nos difficultés en résolution de problèmes : l'organisation en deux niveaux de notre mémoire montre en particulier qu'une fraction vitale du cerveau fonctionne en liaison directe avec ces capteurs. Ces capteurs ont modelé et dirigé le cerveau humain tout au cours de l'évolution. Le système visuel, tout spécialement, joue un rôle primordial. De plus, à force d'imprégnier le cerveau des informations spatio-temporelles il a forcé l'apparition de structures capables de coder efficacement ces types d'informations. Konrad Lorenz n'hésite pas à dire que c'est parce qu'une espèce de grands singes a, un jour, connu une vie plus dure, une chasse plus difficile et a dû alors exécuter au brouillon, dans son espace de représentation et non dans l'espace réel, certaines actions dangereuses pour sa vie, que cette espèce est finalement devenue l'espèce humaine. La prédominance des représentations spatiales était du même coup établie. De fait, l'espace réel nous sert fondamentalement de modèle pour décrire tous les autres rapports en représentation interne. Le langage, en particulier, fait grand usage de données spatiales : « Volez-vous cela clairement ? » Il reste un point important et étonnant : il concerne la taille, estimée expérimentalement de notre mémoire à court terme. Cette taille est remarquablement petite : chez l'homme, la mémoire à court terme ne peut contenir que 7 (plus ou moins 2) informations élémentaires. Il est facile de se convaincre que ce seuil n'est effectivement pas bien élevé. Nous calculons par exemple le produit 7×8 instantanément ; pour le produit 72×8 , le report de la retenue entraîne, on le sent, une petite gestion de la mémoire qui demande un peu plus de temps. Pour effectuer 72×89 ... nous prenons presque tous un papier et un crayon qui font partie de notre mémoire externe habituelle ; si l'usage de ces instruments nous est interdit, l'opération durera vraiment beaucoup plus longtemps : les échanges avec notre mémoire (interne) à long terme sont très longs, nous l'avons vu.

Avec une telle capacité de 7 informations, il n'est pas question de conserver dans notre mémoire à court terme la trace des états antérieurs : toute nouvelle information en détruit une autre. Ce câblage rend ainsi assez **difficile tout retour arrière** dans un raisonnement.

Mais un point reste à éclaircir : la capacité précédente est exprimée en informations élémentaires. Il y a là une notion-clé qu'il est temps de souligner et de définir précisément : ces informations élémentaires correspondent aux blocs d'information que nous sommes capables de reconnaître, de coder, de stocker immédiatement et globalement. **Ils dépendent du domaine et de chaque individu.** Ainsi, si quelqu'un dicte à haute voix une suite de 0 et 1, vous aurez du mal à en mémoriser plus d'une demi-douzaine. Mais les informaticiens sont quelque peu habitués à travailler dans le système octal et disposent alors d'un codage tout fait pour mémoriser en bloc des groupes de trois 0 ou 1 : dès lors, il parviendront, eux, sans effort, à retenir 6 ou 7 chiffres associés à 6 ou 7 blocs, c'est-à-dire en fait une vingtaine d'éléments binaires.

Exemples							Codage	
1 0 1	1 0 0	1 0 1	0 1 1	0 0 0	0 0 1	1 1 1	000 : 0	001 : 1
découpé en :	101	100	101	011	000	001	111	010 : 2 011 : 3
codé en :	5	4	5	3	0	2	7	100 : 4 101 : 5
et on retient : 5 453 027							110 : 6	111 : 7

Un musicien retient quant à lui une mélodie à mesure qu'il l'entend, parce qu'il a, à sa disposition, en mémoire à long terme, un très grand jeu d'informations de niveau élevé. Une oreille peu entraînée aura, à l'inverse, bien du mal à enregistrer les vingt premières notes.

Les informations contenues dans la mémoire à court terme sont donc des noms qui désignent (qui *pointent* sur) les plus gros blocs possibles d'informations correspondants et déjà enregistrés, à un moment donné, chez un individu donné, dans la mémoire à long terme.

6.7 Les études des psychologues sur les joueurs d'échecs

Les travaux de de Groot, de Simon, de Winlkoff ont concerné deux aspects différents de l'analyse échiquier : la perception d'une part, le raisonnement d'autre part. Il faut signaler que de Groot particulièrement a pu étudier et comparer sous ces deux aspects des sujets absolument remarquables. Il a pu en effet analyser les protocoles de six grands maîtres (dont deux champions du monde), quatre maîtres internationaux, deux champions des Pays-Bas et dix experts.

6.7.1 La perception

Les expériences consistent à placer un échiquier correspondant à une position réelle, de milieu de partie en général, devant le sujet. Le mouvement de ses yeux est analysé (enregistré automatiquement dans certains cas par une caméra fixée sur son front). Ensuite, après un temps t_1 , l'échiquier est ôté, on attend un temps t_z , puis on demande au joueur de reconstituer la position qu'il avait sous les yeux. On s'intéresse ici en particulier à l'ordre dans lequel les pièces sont replacées et au nombre de pièces

correctement replacées.

On observe tout d'abord un petit nombre de fixations des yeux sur l'échiquier par seconde : jamais plus de quatre. Chacune de ces fixations autorise technologiquement une vision latérale de deux degrés d'arc et correspond ainsi à la vision d'environ neuf cases adjacentes. De plus, la séquence des fixations révèle, même chez les grands maîtres, des répétitions. Elle montre ainsi que l'échiquier n'est pas balayé selon un ordre géométrique (de gauche à droite, et de bas en haut, par exemple) mais bien, chez tous les sujets, suivant un **ordre logique** qui dépend de la position donnée. Les yeux s'attardent et reviennent sur certaines cases critiques d'où ils repartent en suivant une rangée ou une diagonale. Les cases des rois et les cases adjacentes appartiennent toujours à cette famille. Par contre certaines pièces ou certaines configurations (par exemple chez les blancs : le groupe $f2, g2, h2, Th1$) ne sont « photographiées » qu'une fois. Elles correspondent, à l'évidence, à un bloc d'information élémentaire classique et répertorié chez l'expert. Seules sont plus longues à voir - c'est-à-dire à mémoriser - les configurations les moins familières : notamment les pièces avancées ou mal défendues. Avec un temps d'exposition t_1 de 5 secondes, un temps d'attente t_z de 30 secondes et 24 pièces en moyenne par position, tous les experts reconstituent l'échiquier de manière semblable : dès qu'une pièce est posée toutes les pièces qui sont en relation logique avec elle, par protection directe ou indirecte, par attaque, sont posées. Lors de ce processus, le joueur considère, en cas d'égalité, les pièces aux caractéristiques les plus excentrées d'abord. On ne peut guère espérer des reconstitutions parfaites avec des temps d'exposition aussi réduits. Les grands maîtres y parviennent cependant dans 5 cas sur 14. Les résultats globaux font apparaître une nette différence entre les maîtres et les autres joueurs.

	Grands maîtres	Maîtres	Experts	Joueurs de classe C (*)
Nombre (sur 14) de reconstitutions parfaites	5	4	0	0
Score total sur 264	217	217	158	119

FIG. 6.13 – Evaluation de reconstitutions d'échiquiers (Le score est une fonction construite par De Groot qui tient compte du nombre de pièces sur la bonne case).
(*) Joueurs de classe C : bons joueurs de clubs.

Notons que souvent, le grand maître en particulier a fait beaucoup plus que de reconstituer la position : **il l'a déjà analysée**. A la limite, il ne peut faire autrement, puisque c'est comme cela qu'il la perçoit ; ce sont les lignes de forces de l'action qui ont dirigé ses yeux. Il connaît déjà les coups intéressants, envisageables immédiatement sur la position ou quelques coups plus tard.

Les résultats de De Groot reportés ici concernent les expériences sur des positions réelles. Quand les positions proposées sont tirées au hasard, tous les joueurs, grands maîtres ou amateurs de club, se retrouvent à égalité. Ils sont tous démunis et tous tout à fait incapables de mémoriser l'échiquier en cinq secondes !

Pour l'échiquier de la figure 6.14 (qu'il a pu examiner 5 secondes), le mathématicien, grand maître d'échecs, M. Euwe dit : « D'abord je me souviens que la position des noirs est pourrie », il reconstitue le bloc de pièces autour du roi noir : roi + dame noire + les deux cavaliers noirs + le pion $e6$ + dame blanche et enfin le fou noir $f8$. Le roque blanc est placé d'un coup sans aucun problème. Le temps de pose entre les pièces est chronométré : s'il est inférieur à 2 secondes De Groot considère que les pièces concernées appartenaient au même bloc d'information dans la mémoire. Deux

blocs sont ainsi apparus jusqu'ici chez Euwe. Il note ici que ce genre de position lui rappelle vaguement une partie de Fiohr contre Fine ! (Ceci est invérifiable : il est sûr, en revanche, que ce souvenir a beaucoup aidé la mémorisation de la position soumise à Euwe).

Le doublet de pion $a2, b2$ et par ailleurs les pions $h7, g6$, la tour $h8$ sont ensuite posés ensemble, tandis que toutes les pièces restantes le sont séparément : $e5, Cb5, Td1, Tb4, a7$. Pour ce dernier pion Euwe dit d'ailleurs textuellement : « Je n'ai pas regardé le côté droit des noirs. Mais il doit y avoir un pion noir en $a7$ » (la logique de la partie, les pions $a2$ et $b2$ encore présents corroborent cette information).

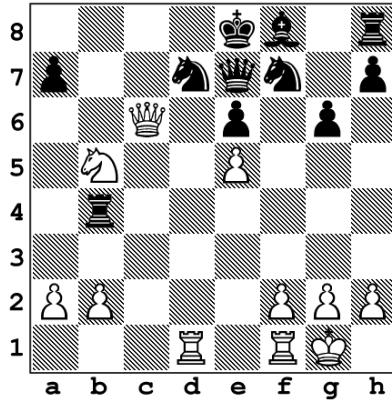


FIG. 6.14 – Position à reconstruire en 5 secondes (aux noirs de jouer)

Euwe continue maintenant « à vider sa mémoire ». Il a vu aussi d'autres faits notables : « L'attaque blanche comprime les noirs, mais j'ai deux pièces en compensation » (Euwe s'identifie à celui qui détient le trait). « Que puis-je sacrifier ? Le cavalier blanc est très fort ; dès lors. $T \times C$ est bon, car cette tour bizarre $b4$ ne fait rien : maintenant je pense que les noirs ont le gain en main, je n'aurais pas dit cela il y a une minute ».

Nous voyons ainsi à quel point la perception est liée au raisonnement, second objet de notre étude.

6.7.2 Le raisonnement

Une situation, issue ici encore d'une partie réelle, est soumise aux experts qui doivent dire quel est, à leur avis, le meilleur coup, ils raisonnent à voix haute. La situation de la figure 6.15 a notamment été soigneusement étudiée par de Groot. Elle est issue d'un match joué par de Groot lui-même. Elle ne comporte pas moins de 56 coups légaux pour les blancs !

Le problème, tous les joueurs en sont convaincus de prime abord, est de nature tactique : Il ne faut pas chercher un plan à long terme, mais bien résoudre la question pressante des nombreuses captures en puissance. Avant d'analyser plus finement les résultats obtenus par les joueurs à l'issue de leur réflexion, donnons déjà les conclusions très nettes qui se dégagent de la forme même de leurs raisonnements.

- a) **La recherche est très sélective.** L'arbre des raisonnements chez les grands maîtres comprend ici de 20 à 76 noeuds. Ce nombre est remarquablement faible

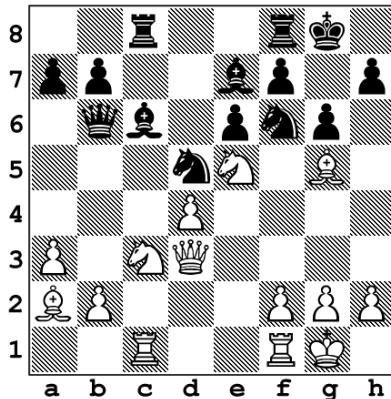


FIG. 6.15 – Etude d'une position (De Groot, 1935)

en face d'une telle explosion combinatoire de coups légaux. Le nombre moyen de coups effectivement envisagés pour les blancs sur la position est inférieur à 5. Keres en particulier envisage trois coups ; Il en étudie deux assez complètement et ce sont les deux meilleurs. Sur l'ensemble des 20 joueurs seuls 22 coups sur 56 possibles ont été considérés.

- b) **La profondeur du raisonnement est faible.** Ce fait est justifié sur plusieurs autres situations. Contrairement à ce que l'on croit souvent, un très grand joueur d'échecs ne raisonne que rarement plus de cinq ou six coups à l'avance (ici par exemple, quatre coups lui suffisent). Aucune différence n'est, de plus, détectée à ce niveau entre les différents groupes de joueurs étudiés.
- c) **Les retours en arrière sont fréquents.** Un coup ou même un épisode entier sont couramment revus plusieurs fois. De Groot distingue pour cette raison trois phases dans l'analyse des joueurs : a) l'exploration ; b) l'élaboration ; c) la preuve. Le joueur d'échec réétudie la position avec un regard différent dans chacun des trois cas. Un coup ami ou ennemi est plusieurs fois retravaillé. L'approfondissement de l'étude est très progressif. Le problème que se pose le joueur est périodiquement redéfini en fonction de l'hypothèse de travail ; celle-ci évolue sans cesse jusqu'à la solution finale. Tous les coups qui, sur l'échiquier, sont sans rapport avec l'hypothèse courante sont, à tout instant, purement ignorés. Le joueur doit, par ailleurs, savoir mesurer la distance à laquelle il se trouve par rapport à l'hypothèse courante et au but qu'il doit atteindre : il lui faut une fonction d'évaluation de la position.

	Nombre de coups envisagés au niveau 1	Nombre total de positions étudiées	Profondeur maximum de l'étude en demi-coups	Temps	Nombre de retours arrières dans l'analyse
Grands maîtres	4.8	35	6.8	9'6"	6.6
Experts	3.4	30.8	6.6	10'8	6.4

FIG. 6.16 – Moyennes de quelques paramètres significatifs lors de l'étude d'une position d'échecs chez plusieurs joueurs.

- d) **L'évaluation utilisée par les joueurs humains est très succincte.** Les phrases qui apparaissent dans les protocoles sont souvent du type : « Les blancs ont maintenant une position gagnante, avec un bon pion passé ». « Les noirs ont

gagné une ligne ouverte ». « La pression de l'attaque est écrasante ».

En outre, très peu de positions sont évaluées dans l'analyse : tant qu'une position stable n'est pas atteinte, tant que le problème que l'expert lui-même se pose à un instant donné n'est pas résolu d'une façon ou d'une autre, il n'est pas question d'évaluer : on retrouve l'indispensable quiescence. L'évaluation enfin, comme toute l'analyse, repose sur des concepts globaux, éloignés de la description banale de l'échiquier. Les notions d'espace (ou de position) ouvert et fermé, de contrôle (d'une case, d'une pièce, d'une zone), de faiblesse et de sûreté réapparaissent ainsi sans cesse. Les coups abstraits comme : « coup de la dame noire, retrait du fou, protection du roi, coup vide, maintenir la pression, se retirer sans perdre le temps » servent **à la fois à juger la position et à construire et développer des buts**.

Venons-en aux détails des raisonnements et à leurs résultats sur la position donnée. Nombreux sont les joueurs qui trouvent le meilleur coup, qui est $F \times Cd5$, ou qui le classent en seconde position. Ce qui est surprenant c'est qu'en général, ce n'est pas pour la bonne raison ! Ce coup est en effet facile à envisager mais l'arbre de la preuve qui le justifie complètement est assez important, ceci entraîne des problèmes de gestion du contexte dans la mémoire à court terme, aussi les experts ont-ils tendance à simplifier l'étude par une évaluation grossière. Ainsi, Alekhine dit très vite : « Je serais très content d'avoir les blancs... $F \times C$ est très fort c'est sûr... vérifions tout ; mais si le temps presse je joue $F \times d5$ ». Le développement de l'arbre lui prend ensuite de nombreuses minutes : se repérer dans de grandes arborescences est difficile pour tout le monde. Même un grand maître peut s'y perdre : Fiohr fait ici par exemple une erreur nette. Considérant, parmi beaucoup d'autres, la variante :

1. $C \times Fc6$ $b7 \times C$
2. $F \times Cd5$ $c6 \times F$
3. $F \times Cf6$ $F \times F$

Il dit que le coup 4. $Cd7$ est une fourchette sur la dame et la tour noires : malheureusement il y a interférence avec un autre échiquier car le cavalier $e5$ n'est plus là. Il ne peut jouer en $d7$!

Assurément, il existe un point commun à toutes les analyses : tous les experts utilisent dans leurs raisonnements une importante connaissance échiquéenne. Ainsi, Alekhine annonce d'emblée que la position lui rappelle une partie récente de Botwinnik ; il dit que l'ouverture doit avoir été celle du gambit dame accepté, que la position correspond au 16^{me} coup ; Il pense être à même de reconstituer logiquement tout le début de la partie. C'est à l'évidence, la connaissance de très nombreuses parties jouées ou étudiées qui permet à Alekhine de situer celle-ci et de la relier dans l'arbre de sa mémoire aux rameaux dont elle est issue.

Cette connaissance, qui autorise la reconstruction historique de la partie, guide du même coup l'analyse : les blancs depuis le début harcèlent les noirs ; ils possèdent *une* position très ouverte et l'avantage de l'attaque. Inversement les noirs doivent résister jusqu'en fin de partie ; dès lors le seul fait de posséder des pions mieux liés leur donne toutes les chances en finale.

C'est précisément ici que jouent les différences de culture entre joueurs d'échecs : une heuristique connue dit en effet qu'un fou est légèrement supérieur à un cavalier, surtout en fin de partie quand le fou se déplace librement sur l'échiquier. Les joueurs moins forts vont ainsi éliminer le coup $F \times d5$, puisqu'il provoque justement la perte d'un fou contre un cavalier. Mais, le grand maître, lui, sait qu'il faut ici chercher l'attaque décisive **avant la fin de partie** ; l'heuristique précédente ne saurait donc être prise

en compte. Les caractéristiques essentielles de la position sont alors pour lui : l'attaque des blancs, qui doit être maintenue ; la force en particulier du cavalier $e5$ qui menace $Cd7$; la relative faiblesse du roque noir : le pion blanc $b2$ non défendu avec incursion possible de la dame noire ; les quatre captures : $C \times d5$, $F \times F6$, $F \times d5$, $C \times f6$. Elles constituent autant de coups blancs plausibles qui doivent être examinés très méthodiquement.

L'ordre dans lequel ces coups vont être considérés est maintenant crucial. En fait, c'est un développement en parallèle qui va d'abord donner des indications, puis la solution, chez les meilleurs joueurs (Cf figure 6.17).

Les effets de ces quatre coups sont en réalité très imbriqués. L'important est de les avoir en tête tous les quatre. Comme dans cette position les coups de fourchettes de cavalier en $d7$ et $e7$ sont forts a priori, l'élimination des cavaliers noirs en conservant les cavaliers blancs est naturelle. L'échange avec le fou $a2$ qui ne joue pas beaucoup est donc bientôt considéré en priorité puisqu'il correspond le mieux à l'esprit de l'attaque sur cette partie. L'analyse de la figure 6.17 est alors effectuée le plus souvent très partiellement, même par les grands maîtres.

Les conclusions sont très nettes. Des travaux des psychologues sur les joueurs d'échecs se dégagent trois idées fortes :

- a) Ce n'est **pas la puissance** de calcul qui fait les grands maîtres, mais leur faculté de **bien choisir les coups à étudier** ;
- b) Le choix des bons coups est lié aux remarquables facultés de **perception** de l'échiquier chez ces joueurs ;
- c) L'acuité de la perception est fonction de **l'étendue des connaissances** et en particulier des **concepts de haut niveau** présents en grand nombre dans le cerveau du joueur et reconnus sur la position considérée.

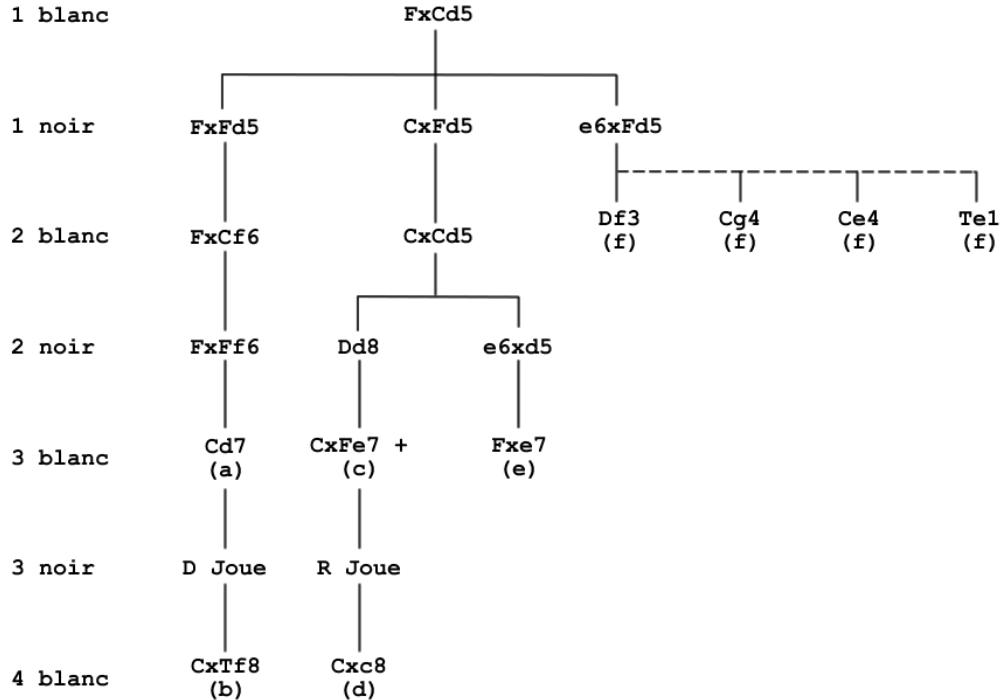
Les psychologues ont, par ailleurs, étudié le comportement humain en résolution de problèmes sur d'autres domaines que les échecs.

Newell et Simon (1982) ont, par exemple, analysé les protocoles d'étudiants travaillant, d'une part, sur des exercices de logique de propositions (cf chapitre 3) et d'autre part sur des problème de crypt-arithmétique (cf chapitre 8). Les stratégies de ces étudiants, les connaissances qu'ils utilisent ont été traduits sous forme de règles de production (cf chapitre 7). Leur démarche générale est ainsi assez fidèlement modélisée.

Mathieu et Thomas (1985) rapportent de plus des travaux effectués sur le jeu de GO, sur l'analyse de circuits électriques, sur des problèmes de mécanique.

Toutes les observations confirment les conclusions du paragraphe précédent : la faculté primordiale est celle qui consiste, en face d'une situation donnée, à détecter les groupements « importants » d'objets (ceux qui partagent le plus grand nombre de relations significatives dans le contexte).

Ceci est lié à l'expérience, aux connaissances du sujet et à la **qualité de la structuration** de celles-ci.



- (a) Fourchette dame, tour ;
- (b) Gain d'une qualité finalement ;
- (c) Fourchette roi. tour ;
- (d) Gain pièce + qualité ;
- (e) Gain d'une pièce ; si et si 2... $F \times Cd5$. 3 $F \times Cf6$;
- (f) Le cavalier noir $f6$ est maintenant très faible et les blancs gardent une excellente attaque ;
- (g) Le fou noir $e7$ est sous la menace d'une attaque à la découverte.

Le fait qu'il existe quatre coups blancs après la réponse $e6 \times F$ suffit pour considérer cette riposte noire comme insatisfaisante. Le gain des blancs, dans toutes les variantes, se démontre d'ailleurs facilement ; mais la pression est déjà assez élevée pour que tous les grands maîtres se contentent ici de cette profondeur d'analyse.

FIG. 6.17 – Arbre de preuve du coup $F \times Cd5$ dans la position A (de Groot 65)

6.8 ROBIN : Un programme d'échec intelligent

Nous avons rencontré, chapitre 6.6.1, une première famille de programmes pour jouer aux échecs. Leurs caractéristiques étaient très éloignées de celles d'un joueur humain. Comme l'homme a, par ailleurs, dans ce domaine difficile, de meilleures performances que ces programmes, il devenait capital pour les chercheurs en intelligence artificielle de **comprendre** comment l'homme s'y prenait pour parvenir à ces résultats supérieurs en développant un arbre de recherche **mille ou dix mille fois plus petit que les ordinateurs**. Le mieux était d'écrire effectivement un programme dont le principe serait autre que l'énumération systématique des coups. C'est à un tel programme que

Jacques Pitrat, chercheur au CNRS à Paris, s'est attaqué dès 1972.

La remarque essentielle est la suivante : s'il existe une telle différence entre les arbres de recherches des hommes et ceux des programmes, c'est **qu'il ne s'agit pas des mêmes arbres !**

Les études de Binet, de Groot, Simon l'ont montré l'expert perçoit rapidement les caractéristiques marquantes d'une position. Il « voit », littéralement, les coups à envisager en priorité. Ainsi, le bon joueur commence sa recherche avec une **analyse extrêmement approfondie de la situation initiale** : un vocabulaire très précis lui permet d'ailleurs toujours de décrire finement les potentialités de la situation, les points forts, les faiblesses de chaque coup sont ainsi perçues et énoncées. Toute la recherche du meilleur coup sera ensuite guidée par cette analyse initiale, aussi bien **en largeur** - par restriction aux seuls coups pertinents en fonction de cet examen. - **qu'en profondeur** -par descente jusqu'à stabilisation des déséquilibres créés par des coups intermédiaires.

En second lieu, il apparaît une autre différence notable avec les humains : l'étude des coups dans le cerveau du joueur, contrairement aux programmes de la première famille, **ne se fait pas dans l'ordre réel du jeu sur l'échiquier** : des coups impossibles sont envisagés, un coup à plusieurs temps d'avance est considéré d'abord, en un mot des *plans* sont construits, développés, remis en cause, abandonnés, repris et complétés jusqu'au coup final.

C'est alors bien d'une **arborescence de plans** qu'il est question et non plus d'une énorme et aveugle **arborescence de coups**. La plupart des coups trouvés par le programme de Jacques Pitrat échappent d'ailleurs complètement aux autres programmes... et quelquefois aux spécialistes eux-mêmes ! On tient là **une deuxième famille de programmes de jeux**, étrangère à la première et seule intéressante finalement en intelligence artificielle, car ses principes mêmes sont transposables à de nombreuses autres activités humaines.

Donnons, tout d'abord, un exemple de comportement du programme de Jacques Pitrat. Nous décrirons ensuite ce programme, dont le nom est ROBIN, plus finement.

6.8.1 Un exemple d'analyse par ROBIN :

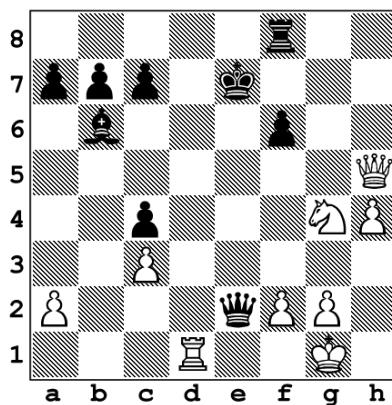


FIG. 6.18 – E42 Trait aux blancs

Pour jouer un coup dans cette position, le programme commence par entreprendre **une analyse fine de la position**. Il construit en particulier des **plans** possibles pour son camp. Parmi ceux-ci, on trouve :

- B1) l'attaque directe du roi ennemi, début par $Dh7+$
- ou bien B2) l'attaque double (= *clouage*) roi + dame par la tour en $e1$
- ou bien B3) la prise de cette dame noire non protégée, $D \times D$

Ces plans sont des actions - en réalité des suites de coups élémentaires - prévus à l'avance, en fonction d'un **but à atteindre**, ici prendre le roi ou la dame noire. Ces buts sont souvent **ambitieux**, quelquefois **irréalisables** ! Mais, en développant ces plans, le programme va s'efforcer de rendre possibles les coups qui ne le sont pas initialement, créer des menaces, faire bouger la défense adverse et, finalement, sinon gagner le but initial, du moins en tirer bénéfice.

Notons que ce **développement à partir des buts au jeu à rebours** par rapport au jeu réel des coups sur l'échiquier : le programme **remonte le raisonnement** à partir du but pour en déduire, au premier niveau, le coup correct. Celui-ci peut être tout à fait inattendu et non naturel pour qui n'a pas le but visé en tête !

Ici le programme développe B1) et B2). Mais, c'est finalement B3) qui va réussir, compte tenu des possibilités de ripostes noires, prises en compte à mesure (on ne donne ici que le **seul plan** qui réussit finalement ; de nombreuses autres variantes sont en réalité également étudiées par le système) :

- a) Le programme envisage de prendre la dame $e2$ par $D \times D$.
- b) Il faut donc ôter le cavalier blanc $g4$.
- c) Mais la riposte noire est $D \times Dh5$.
- d) Il faut ôter le cavalier en attaquant une pièce ennemie d'importance au moins égale à la dame : il n'y a que le roi.
- e) Le sous-but « mettre échec au roi $e7$ par le cavalier » est créé.
- f) Aucun échec direct n'est possible, les seules cases pertinentes pour le cavalier sont $e5$ et $f6$, vue la position du Roi.
- g) Un nouveau sous-but est de déplacer le Roi noir.
- h) Seule la tour $d1$ peut être de quelque utilité pour accomplir ce travail (les coups de la Dame sont en effet rejetés car le coup $D \times D$ doit rester possible.)
- i) La case $e1$ n'est pas protégée (Cf. plan B2).
- j) Il reste donc $d7$ avec échec au roi.
- k) D'où la clé $1-Td7+!$ avec la variante principale : $1-Td7+, R \times T, 2-Cf6+$ (surtout pas : $Ce5+$ car : $D \times C$ et s'échappe). $T \times C$ et $3-D \times De2$; le programme vérifie alors que $3... F \times f2+$ ou $3... T \times f2$ ne donne rien aux noirs.
- l) Enfin il subsiste la possibilité pour le Roi de ne pas prendre le cadeau empoisonné $Td7 : 1... Re6$ (seule case autorisée). Mais c'est pire alors car suit le coup : $2-Dd5$ mat.

Certes, cette position était, à la limite, à la portée des anciens programmes (*profmax = 7demi – coups*). Mais elle illustre bien la possibilité d'engendrer le meilleur coup en ne construisant qu'un **arbre extrêmement maigre** : 305 coups ont été considérés en tout et pour tout contre environ 40^7 soit 16.10^{10} pour un programme énumératif. ROBIN peut ainsi, sur des positions complexes, comme nous le verrons, descendre à une profondeur de **vingt demi-coups et plus**.

6.8.2 Différents types de plans aux échecs

Toute l'analyse de la position initiale est axée dans ce programme vers la détection de configurations favorables. C'est elles qui donnent naissance à des **plans**, c'est-à-dire à des suites linéaires de buts (ou coups généralisés) qui se terminent, au bout du compte, par un **gain matériel**. Le programme s'intéresse donc uniquement aux combinaisons de cette nature. Il existe pour ROBIN quatre types de plans différents dont le plus simple est ainsi la prise pure et simple d'une pièce ennemie.

Plans de type 1 : capture simple.

Pour chaque pièce, le programme dénombre - ce qui est parfaitement algorithmique et rapide - le nombre et la valeur des pièces qui l'attaquent et la protègent. Les attaques ou protections potentielles, « à la découverte », c'est-à-dire masquées par une pièce amie, sont également retenues ici. Le bilan est alors effectué. Toute pièce, quantitativement ou qualitativement moins ou autant défendue qu'elle n'est attaquée, est réputée en position de capture simple et donne naissance à un plan.

Dans la position précédente (Position E42) nous avions vu que la dame noire était virtuellement en prise. Pour les noirs, la tour blanche *d1* et le pion blanc *a2* sont directement en prise ; le pion *f2* et le cavalier *g4* peuvent être pris soit en les attaquant une fois de plus, soit en faisant partir au moins un de leurs défenseurs.

Plans de type 2 : attaque double.

Il existe évidemment un cas plus prometteur que le précédent, il correspond à une situation dans laquelle une **même** pièce amie attaque simultanément plusieurs pièces ennemis (en général deux). C'est un cas favorable puisque l'ennemi, qui n'a droit qu'à un seul coup, ne pourra souvent sauver qu'une des pièces ainsi menacées : c'est l'attaque double dont un cas particulier est le clouage (*De2* serait clouée - et donc perdue - dans le diagramme précédent si l'on ajoutait une tour en *e1*). Le Roi ennemi peut toujours être l'une des pièces soumises à une attaque double : il n'est en effet jamais directement défendable.

La question posée ici par le programme à son générateur de coup est donc : « Existe-t-il une case où peut se rendre une pièce amie et de laquelle elle attaquerait deux pièces ennemis, toutes deux en position de capture simple ? ». En outre, quelques libertés sont données au générateur :

- La case cible peut être déjà occupée : un sous-but sera alors de chasser la pièce.
- Une case intermédiaire peut également être initialement occupée : il faudra donc la libérer.

Exemple :

Dans cette position M58 (fig 6.19), l'analyse des captures simples relève les captures du Roi blanc *e1* et aussi, en particulier, celle du pion *a2* : Il existe donc une possibilité d'attaque double de ces deux pièces faibles par la dame noire, via la case *e6*. Celle-ci est occupée, il faut d'abord ôter le pion : ensuite un autre pion - *d5* - masque *a2* : il faudra penser à l'ôter également.

Plans de type 3 : attaque du Roi.

Il faut bien à un moment ou à un autre, quand on joue aux échecs, s'intéresser au rôle particulier du roi parmi les autres pièces : s'il est pris tout est perdu. C'est donc une source éventuelle riche pour les plans ennemis.

Tous les coups donnant échec sont des débuts de plans, en tous cas dès que la pièce donnant échec n'est pas attaquée ou bien, si elle l'est, si le Roi ennemi n'a d'autre coup que de prendre celle-ci. Ainsi, dans la position (E42), le coup : *Td7+ ne* donne **pas** initialement naissance à un plan, au contraire du coup : *Df7+*. Car, dans ce second

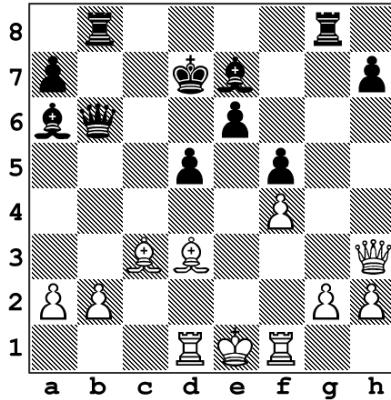


FIG. 6.19 – M58 Trait aux noirs

cas, il n'y a plus de case de fuite pour le roi : l'attaque est a priori beaucoup plus forte. Mais il faut, pour la compléter, et c'est le développement du plan qui dira si c'est possible, d'une part, ôter la tour noire $f8$ et, d'autre part, protéger la dame blanche en $f7$.

Enfin il existe sur l'échiquier un autre type de pièce qui joue un rôle particulier : c'est le pion. Arrivé en huitième rangée. Il se change en effet en pièce à sa guise.

Plans de type 4 : promotion.

Comme il ne convient pas de créer un plan « aller à dame », pour n'importe quel pion, le programme détecte, en milieu de partie, les pions passés et donnera naissance à tel plan pour les seuls pions déjà parvenus en sixième ou en septième rangée. Le plan le plus simple sera alors de pousser le pion (en vérifiant que le bilan reste toujours favorable). Si une pièce bloque la marche, ROBIN envisage, en outre :

- soit d'ôter cette pièce ;
- soit de faire venir sur une case voisine une pièce ennemie en exploitant la possibilité de prise en diagonale par le pion.

Tous ces plans initiaux sont engendrés parallèlement par le programme **pour les deux camps**. De plus, pour limiter la recherche à des plans qui possèdent a priori une bonne chance de fonctionner **contre toute défense** (le programme ne conserve que les plans qui n'exigent initialement **qu'au plus deux modifications** de cases intermédiaires).

6.8.3 Langage d'expression et d'exécution des plans :

Le langage de plans

De même qu'il est commode de coder habituellement les coups : $Th7$, $b \times c$ ou $o - o$ pour pouvoir les engendrer et les manipuler. Il est utile, pour un programme qui gère une arborescence de plans, d'avoir un langage pour exprimer ces actions ou « supercoups » qui apparaissent dans ces plans.

Une action typique est la demande de *modification* d'une case que nous avons déjà rencontrée. Cette modification est codée : une case est. à un moment quelconque, vide V , ou bien amie A , ou bien ennemie E . Dans ces deux derniers cas la pièce occupante est indiquée. Le codage et la signification des actions dans les plans sont donnés dans les exemples qui suivent :

$DE.e2 \rightarrow V$	pour : "ôter la dame ennemie de la case e2"
$E.a4 \rightarrow \neg E$	pour : "transformer l'occupation ennemie de a4 en vide ou en amie"
$V.a8 \rightarrow E \geq C$	pour : "faire venir en a8 initialement vide une pièce ennemie de valeur supérieure à un cavalier"
$CA.g4 \rightarrow V^*$	pour : "ôter le cavalier ami de g4". La présence de l'étoile précise :"effectuer cette action en conservant l'attaque"

Dès lors, le plan lié à la double attaque de la dame noire en M58 s'écrit :

$A.d5 \rightarrow V^*$	"ôter le pion noir d5. Garder l'attaque."
$A.e6 \rightarrow V^*$	"idem, pour e6"
$DA.b6 \rightarrow e6$	"placer la dame amie b6 en e6"
$DA.e6 \rightarrow e1 \text{ ou } a2$	"concrétisation d'attaque double"

Développements de plans

Le programme étudie ensuite chacun des plans les uns après les autres. Chaque action est alors précisée par indication de **l'un** des coups qui peut la réaliser. Un même plan initial donne ainsi plusieurs plans dérivés, Les actions qui ne sont pas directement possibles sont par ailleurs complétées, Par exemple, le programme sait que **pour ôter une pièce :**

- **si elle est amie, il suffit de la déplacer ;**
 $CA.g4 \rightarrow V^*$ donne d'abord ainsi dans la position E42 : $CA.g4 \rightarrow h2$ ou $f2$ ou $e3$
 $e5$ ou $f6$ ou $h6$
- **si elle est ennemie : - la prendre.**
ou - **l'attaquer**
ou - **lui donner quelque chose à prendre.**

C'est ainsi que dans la position E42 comme aucun des coups de cavalier ci-dessus ne satisfait la condition imposée par l'étoile (garder l'initiative), le programme a dû créer l'action nouvelle : $RE.e7 \rightarrow V^*$ puisque avoir l'initiative c'est ici « ôter » le roi.

Nous venons de voir, comme le programme, que le cavalier ne peut ni prendre, ni attaquer, ni être pris.

Il reste les coups de la tour d1. Et $RE.e7 \rightarrow V^*$ devient :

- $TA.d1 \rightarrow T \times R^*$, prendre le roi
- ou $TA.d1 \rightarrow T+^*$, l'attaquer
- ou $TA.d1 \rightarrow Td6$, $d7$ ou $d8$ lui donner quelquechose à prendre

La première de ces nouvelles tâches ne correspond à aucun coup sur l'échiquier; le sous-plan associé est abandonné. La suite correspond à un coup unique $Td7$ (qui se trouve satisfaire aussi la troisième puisqu'alors $R \times T$ est possible).

L'étude de ce plan est donc poursuivie; actuellement :

1. $Td7 R \times d7$ ou R bouge
2. C donne échec et 3. $D \times D$.

Pour tout type d'action le programme dispose d'un répertoire de sous-but à même d'accomplir celle-ci.

Donnons un autre exemple de ceci. Pour détruire une prise (ce qui constitue un cas fréquent d'échec local d'un plan), il faut :

- déplacer la pièce qui prend,
- ou bien interposer une autre pièce (amie ou ennemie).

Dans la position M58, un plan possible est lié à $De3+$; mais la riposte $D \times D$ est immédiate.

« L'interposition » suggère :

$V.g3 \rightarrow \neg V$ (faire venir quelquechose en $g3$)

$DA.b6 \rightarrow e3+$

« Quelque chose » signifie en particulier un pion ennemi; un pion ennemi peut se déplacer si on lui donne quelque chose et, puisqu'il prend alors en diagonale, c'est le pion $h2$ qui est intéressant. $V.g3 \rightarrow V$ est donc réécrit : $PE.h3 \rightarrow g2$. Sur l'échiquier réel, une pièce peut venir se poster en $g3$, sans aucune raison, il est vrai, sauf par rapport au plan considéré, c'est la tour $g8$.

Maintenant : $1-Tg3 h \times g3$ donne : $2-De3+ Fe2$.

Ce dernier coup est trouvé par le programme - qui fabrique continuellement aussi les plans pour l'ennemi - à l'aide de la même idée d'interposition. Mais la prise simple : $D \times e2 \neq$ est licite pour les noirs, qui trouvent donc le gain dans cette variante. Un autre plan blanc, pour ôter la tour $g2$ qui menace : $T \times O$, est bien entendu : $D \times T$. Mais cette fois, la dame blanche, qui doit partir toujours en vue de la réussite du plan initial, a changé de case.

Un coup légal peut la chasser à nouveau, c'est $Fh4$, que le programme trouve immédiatement à l'aide de son répertoire de sous-but. Les noirs gagnent la dame blanche. Dans cette recherche, de très nombreux plans sont rejettés par le programme. Il peut en effet arriver :

- a) qu'un coup vital pour le plan soit désormais impossible (par prise de pièce(s) par exemple);
- b) que la balance du matériel soit fortement défavorable (ou encore inférieure à ce qui est demandé en donnée), même si le plan courant réussissait.
- c) que le plan soit devenu logiquement impossible par présence d'une contradiction (comme : $x y \rightarrow V$ et $x y \rightarrow TA$).
- d) qu'une contrainte soit impossible à saisir (par absence de coup par exemple).
- e) qu'un plan soit rejeté par une simulation optimiste : même si tout se passe au mieux, le bilan total en matériel est négatif.

Enfin, comme les plans sont complétés et modifiés sans cesse, il est nécessaire pour le programme de **garder trace de l'idée de base du plan**. Ceci peut se faire par une action particulière qui exprime qu'il faut à un moment donné **vérifier une condition** : par exemple, on a voulu libérer une case pour rendre possible un déplacement ; il faut vérifier plus tard, alors que des coups ont été ajoutés en amont et après des interférences possibles, que la case prévue est bien vide au moment idoine.

6.8.4 L'optimisation de la défense : la notion de coup dangereux

En réalité le programme, à tout instant, considère le jeu d'un seul joueur : **celui qui perd** dans le plan en cours d'étude. A chaque étape, en effet, un joueur va vers un gain que l'autre joueur essaye de contrer.

Pour trouver une réplique correcte, M cherche lui-même des plans. Ce faisant, il peut reprendre l'avantage ; c'est au jeu de son adversaire que le programme s'intéressera

alors à nouveau. Bien sûr, il peut arriver qu'un même joueur soit perdant plusieurs coups de suite. C'est même de cette façon que l'on parvient à la fin de l'étude : le perdant a tout essayé et n'a rien trouvé pour se sauver.

Le développement de l'arbre de recherche lui-même semble ainsi très proche de celui que construisent les humains aux échecs : le chemin le plus prometteur est étudié d'abord. On regarde alors comment l'adversaire peut **parer au pire**. Si une réfutation est trouvée, une autre ligne de jeu à plus haut niveau est envisagée. Si elle se trouve elle-même contrée, la première est poussée plus à fond et ainsi de suite.

Tout le jeu tient donc en la découverte et la réfutation de « *coups dangereux* ». Ces coups vont recevoir une définition précise dans ce paragraphe. Eliminer le pire consiste alors à détruire de tels coups ou bien à en corriger les effets au moyen d'autres coups dangereux. On s'attaque d'abord aux contres des coups qui, en cas de succès, laissent le moins d'autres coups dangereux à contrer.

Pour trouver un plan qui réfute un plan ennemi, il faut **réguter tous les coups dangereux du plan ennemi**. L'idée, pour atteindre ce but, est de réagir différemment suivant le type des coups dangereux rencontrés. Ces coups peuvent en effet être classés en quatre types différents et une méthode appropriée de riposte va être donnée dans chacun des cas.

La définition d'un coup dangereux est, pour le programme, liée à **l'évaluation** et à la procédure de remontée par **minmax**. L'évaluation d'une feuille est prise égale à la seule **balance du matériel**.

Supposons que le programme recherche pour le joueur J un plan qui lui rapporte $v(J)$. Un coup **dangereux** est alors tout coup ennemi qui au moment de l'étude, conduit à une évaluation inférieure à la valeur attendue $v(J)$.

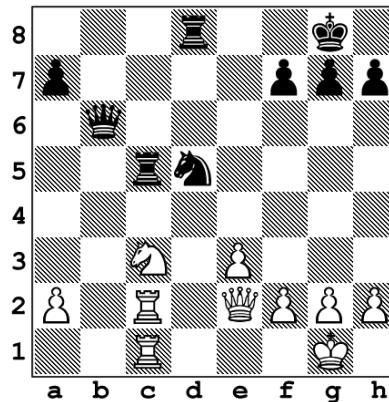


FIG. 6.20 – M39 Trait aux noirs

Exemple M39 : Un des plans engendrés par le programme est lié à l'avance de la tour noire $d8$ en $d1$ où elle contrôle toutes les cases de fuite du roi blanc. Ce plan s'écrit :

$$CA.d5 \rightarrow V \text{ (ôter le cavalier)}$$

$$TA.d8 \rightarrow d1 \text{ (venir en } d1\text{)}$$

$$TA.d1 \rightarrow T \times R$$

La balance du matériel est bonne pour les noirs (prise du roi) le coup $T \times R$ est dangereux pour les blancs et le programme, qui se met donc du côté des blancs, regarde comment le supprimer. Il va trouver immédiatement $D \times Td1$, $T \times Td1$. Mais le coup $CA.d5 \rightarrow V$ qui peut être $C \times Cc3$ était lui aussi dangereux pour les blancs. L'équilibre

est cependant, là aussi, vite rétabli par $T \times Cc3$.

Il n'existe alors plus de coup dangereux pour les blancs : toutes les menaces viennent d'être annihilées. Le programme reconsidère donc le jeu des noirs.

Les trois coups blancs qui viennent d'être construits sont, en effet, autant de coups dangereux pour les noirs.

Décrivons maintenant comment le programme classe les coups dangereux et en détruit les ripostes possibles.

Les quatre classes de ripostes à un coup dangereux sont constituées des **nouvelles captures, des destructions logiques du coup dangereux, des interférences, des plans locaux**.

1. - Nouvelles captures : Tout coup aux échecs, à part le roque et la prise en passant, modifie exactement les contenus de deux cases. Le dernier coup ennemi et le coup ami qui l'a précédé ont modifié quatre cases. Le programme regarde de façon systématique si ces quatre modifications ont créé la possibilité de captures (de valeurs bien entendu supérieures à celle du coup dangereux qu'on cherche à compenser).

Dans une position standard aux échecs, cette première classe de réfutations permet, très vite, d'approfondir les premiers plans.

2. - Destructions logiques : Il s'agit ici de faire tout simplement disparaître le coup dangereux de la liste des coups légaux pour l'adversaire.

Il est possible d'agir soit sur la case de départ, soit sur une case intermédiaire, soit sur la case d'arrivée, de ce coup dangereux. Faire partir la pièce dangereuse Q de sa case de départ d correspond à l'action : $QE.d \rightarrow \neg E$. Inversement, si la case d'arrivée a contient une pièce amie P , l'action : $PA.a \rightarrow \neg A$ interdit le coup dangereux ennemi. La dernière possibilité est d'interposer sur toutes les cases utiles u situées sur le trajet de d à a une pièce quelconque, soit : $V.u \rightarrow \neg V$.

3. - Interférences : Cette classe de réfutation considère l'**attaque** comme une des défenses possibles...

Le programme laisse le camp ennemi exécuter son plan mais profite des modifications introduites par son adversaire sur le terrain pour mettre en place un contre-plan que lui-même déclenchera plus tard.

Ce type de défense est très trompeur : le programme de J. Pitrat a dû ainsi « corriger » quelques analyses de joueurs d'échecs célèbres (Cf. la position T149 dans les exemples qui suivent).

La possibilité de tenir compte de ces interférences tient à deux raisons :

- a) la première est que les actions des plans ne sont pas nécessairement légales sur l'échiquier actuel : elles sont donc insensibles à certains changements et les plans prévus restent en particulier valables pendant plusieurs coups de suite.
- b) la seconde est qu'initialement tous les plans de l'adversaire sont aussi engendrés par le programme. En cas de danger pour l'adversaire il est donc facile pour le programme, qui s'est mis à sa place, de les essayer.

Dès que la première action d'un plan défensif devient possible, elle est ainsi placée avec le plan associé dans la liste des ripostes envisageables au dernier coup adverse.

4. - Plans locaux : il s'agit ici de construire une **attaque indirecte** de la pièce qui peut effectuer le coup dangereux : attaque double, menace de mat, prise après retrait d'un défenseur.

Toute cette analyse est très similaire à celle qui est faite au départ pour toutes les pièces sur tout l'échiquier. Cette analyse est tout simplement limitée ici à une case.

6.8.5 Résultats

Ainsi s'achève la description, dans ses éléments principaux, du programme d'échecs ROBIN de J. Pitrat. Vous voyez à quel point il est **différent des programmes aveugles** bâties autour de l' α - β . Il raisonne véritablement en poursuivant une idée, un plan qu'il a lui-même construit.

Il n'est **aucunement limité a priori en profondeur ou en largeur** dans sa recherche. L'arborescence qu'il développe est une arborescence de plans, c'est-à-dire de suite d'actions ou « méta-coups » susceptibles de donner naissance à des familles de coups légaux. Cette arborescence est donc a priori remarquablement plus réduite que celle des coups réels. En outre, seules les actions pertinentes pour chaque plan étudié sont développées. **La recherche n'a pas lieu selon la suite des coups au cours du temps dans le jeu réel mais selon la séquence logique des actions du plan.**

Ses résultats sont largement à la hauteur des résultats humains sur des positions d'échecs tout à fait réelles. Ce programme a trouvé des combinaisons qui avaient échappées à des joueurs remarquables comme Euwe, Lasker ou Smysiov, à des analystes comme du Mont. Il a démontré que des combinaisons trouvées par du Mont ou Tarrasch ne tenaient pas (dans une variante l'ennemi s'échappe). **Examinons quelques analyses du programme. Exemple M42 :**

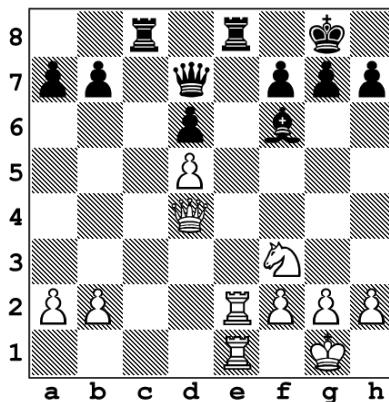


FIG. 6.21 – M42 Trait aux blancs

Cette position est très difficile. Le programme va montrer en descendant finalement à une **profondeur correspondant à 23 demi-coups** que les blancs, s'ils jouent au mieux, gagnent au minimum une tour noire, contre toute défense.

L'idée qui conduit au plan gagnant est l'attaque directe du roi noir par : $T \times Te8+$, en effet ce roi n'a pas de case de fuite. Ce plan va conduire au succès, car la dame noire $d7$, qui doit protéger à la fois $c8$ et $e8$, est surchargée. Le plan P1 est ainsi :

P1	$T \times T.e8$
	$T \times R.g8$

Le dernier coup de P1 est - on ne peut plus !... - dangereux pour les noirs ; ils le contrent par la prise de la pièce qui joue ce coup dangereux. Deux pièces noires sont prêtes à reprendre $Te8$, ce sont la dame et la tour noires. Pour les blancs, ces deux coups sont donc à détruire. En particulier, le premier plan est changé en :

P2	$DE.d7 \rightarrow \neg DE$ $TE.c8 \rightarrow \neg TE$ suivi de P1
----	---

Pour ôter la dame noire de la case clé, parmi d'autres stratégies, nous avons vu que le programme savait proposer une pièce qui, en se sacrifiant, éloignait l'un des défenseurs. Ceci donne ici, parmi plusieurs autres plans, le plan P3.

P3	$DA.d4 \rightarrow g4$ $\neg DE.e7$ (état à vérifier : il peut être obtenu soit par $DA \times De7$, soit par $DE \times Dg4$) suivi de P2
----	---

Arrivé, à ce point le programme voit qu'il reste encore un coup dangereux noir : en effet, la dame noire peut encore garder le contrôle de la case $e7$ sans se mettre en position de prise. Il lui faut et il lui suffit pour cela de se placer en $b5$.

Ce nouveau coup de défense doit donc être détruit ; d'où, pour les blancs, le nouveau plan qui est sous-cas du plan P3 :

P4	$DE.b5 \rightarrow \neg DE$ suivi de P2
----	--

En outre il y a, dans le plan sous-jacent P2, un autre coup dangereux depuis le début c'est : $T \times e8$. Or, ces deux coups sont ici détruits chacun séparément par le même coup blanc : $Dc4$.

Deux suites sont possibles : $D \times Db5$ ou bien $D \times Tc8$.

P5	$DA.g4 \rightarrow c4$ $DA \times Db5$ $\neg DE.b5$ suivi de P2
----	--

C'est en poursuivant encore cette dame noire, avec toujours la même idée, que les blancs parviennent effectivement à faire finalement aboutir ce plan, dont la variante principale est :

- 1 - $Dg4 \quad Db5$
- 2 - $Dc4 \quad Dd7$
- 3 - $Dc7 \quad Db5$
- 4 - $a4 \quad D \times a4$
- 5 - $Te4 \quad Db5$
- 6 - $Db7 \quad$ la dame noire est prise car sinon $Te8 \neq$

Mais plusieurs autres coups sont possibles pour les noirs par exemple en 5 : ... $T \times e4$ suivi de 6 : $D \times c8$, $Te8$: mais 7 : $D \times e8+$.

Au total, pour prouver que le premier coup est le bon et qu'il gagne la dame noire quoi qu'il arrive, il faut ici, de toute façon, descendre à la profondeur de 23 demi-coups et, dans l'une des variantes, à la profondeur 11, les noirs n'ont pas moins de 26 ripostes à leur disposition.

On voit donc que le programme a un jeu parfait et qu'il le démontre. Ce dernier point n'est pas nécessaire au joueur humain ; Il lui suffit en fait de montrer que le coup qu'il joue ne perd pas et gagne dans la plupart des variantes. Dans cette position M42, la démonstration de la correction de 1. $Dg4$ n'est d'ailleurs pas à notre portée : le pro-

gramme a calculé qu'il fallait pour cela au minimum (meilleur coup blanc trouvé tout de suite à tous les niveaux) étudier 2454 variantes...

Exemple B5 :

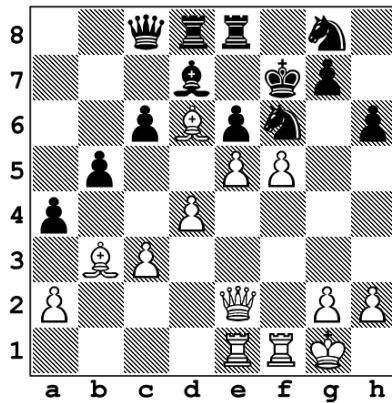


FIG. 6.22 – B5 Trait aux blancs (Berliner)

Berliner proposait cette situation dans sa thèse pour montrer à quel point étaient nécessaires, quelquefois, les arbres très profonds.

Le programme de Berliner trouve le coup correct de la manière qui suit : le roi noir est en danger puisque toutes les cases qui l'entourent lui sont inaccessibles ; le coup d'échec $Dh5$ peut être fatal. Un programme d'énumération systématique rejette rapidement ce coup après un ou deux échanges car il ne « voit » pas comment récupérer la dame blanche après $C \times Dh5$.

Au contraire ROBIN cherche pour les blancs une façon de contrer le coup dangereux $C \times D$.

Une analyse locale de la case où se trouve la pièce qui rend ce coup possible, met en évidence une contre-attaque - ici un clouage - de cette même pièce. La tour $f1$ est susceptible de jouer ce rôle sous réserve d'ajouter au plan l'action : faire disparaître ce pion (ami) $f5$.

Mais comme le coup à prise $f5 \times e6$ est légal, c'est chose facile. Cependant le programme voit ici que le jeu du coup $f5 \times e6$, dès le début, autorise le coup noir $Te8 \times e6$ qui est dangereux pour les blancs, et est finalement imparable puisqu'il ouvre au roi noir une case de fuite salutaire !

$Dh5$ suivi de $f5 \times e6$ reste jouable puisque ce dernier coup donne échec (en fait échec double, après $C \times D$) à un roi qui n'a toujours qu'une case de fuite ; ce plan, parallèlement à plusieurs autres, est développé, mais le programme sait qu'il doit récupérer une dame au moins après le premier échange :

1 - $Dh5 + C \times h5$

La procédure d'analyse limitée sur les cases occupées par le roi permet alors d'engendrer la suite de coups :

- 2 - $f5 \times e6++ Rg6$
 3 - $Fc2+ Rg5$
 4 - $Tf5+ Rg6$

D'autres plans ont été étudiés.

Mais celui-ci conduit au coup 5

avec toute la force du nouvel

échec double :

La tour $g6$ protégée par le fou

$c2$ et le roi doit maintenant

avancer vers l'ennemi !

Et l'autre tour blanche entre en

scène :

- 5 - $Tf6++ Rg5$

Tous les coups noirs jusqu'ici

- 6 - $Tg6+ Rh4$

- 7 - $Te4+ Cf4$

- 8 - $T \times f4 Rh5$

sont forcés : c'est pourquoi les blancs déroulent leur plan fermement, à la poursuite d'un roi éloigné de ses troupes, aucune pièce noire ne peut jouer.

La suite est facile : le coup blanc donnant l'estocade ultime : $Th4+$ est prématué à cause du coup dangereux : $R \times Th4$. Il faut donc interdire ce coup auparavant par la simple poussée $g3$ d'où 9 : $g3$, ad libitum. Aucune pièce noire ne peut entrer dans le jeu et le roi noir est déjà immobilisé 10 : $Th4 \neq$.

Exemple T149 :

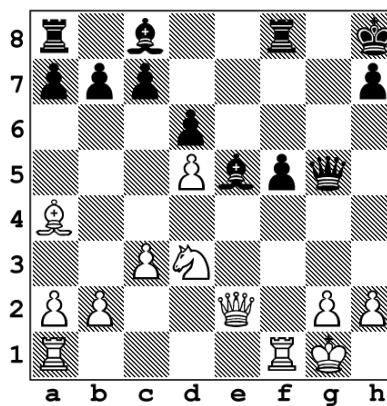


FIG. 6.23 – T149 Trait aux noirs

Tarrasch dans son livre indique le gain d'un pion par

1 : ... $F \times h2 +$, 2 : $R \times h2 Dh4 +$, 3 : $Rg1$, suivi de $D \times Fa4$

Cette analyse repose sur le plan d'attaque double sur $h2$ et $Fa4$. Mais le programme qui engendre aussi les plans des blancs voit une interférence possible avec le plan blanc d'attaque double sur $c7$ et $Tf8$ après libération de la case $e5$, libération qui se produit justement au premier coup du plan noir.

Le plan blanc s'enclenche donc bien après celui donné ci-dessus pour les noirs avec :

4 - $De7!$

Une des ripostes jouables pour les noirs est la protection simultanée des deux pièces à l'aide du coup 4... $Fd7$. Mais ce plan d'attaque du fou ou de la dame, qui, seuls, se protègent mutuellement, et un autre sur le pion $h7$ mal défendu, donnent : $Tf4$, puis : $Th4$, avec un avantage des blancs qui réduit à néant le pion pris au premier temps. ROBIN montre ainsi que cette analyse de Tarrasch était incomplète.

Chapitre 7

Les systèmes experts

Introduction

Les ordinateurs seront de moins en moins utilisés pour le traitement d'informations numériques. La plupart des logiciels étaient jusqu'ici des algorithmes qui travaillaient avec une information complète.

Ce chapitre présente une famille de programmes - les systèmes experts - qui se veulent des aides au raisonnement humain dans divers domaines bien définis. Leurs caractéristiques essentielles sont : la manipulation de connaissances symboliques, le raisonnement dans un univers incertain et incomplet, la communication très naturelle avec l'homme dans un langage modulaire non procédural.

La première partie présente leur conception formelle et diverses réalisations.

Dans la seconde partie, ces systèmes seront discutés, les divers modes de représentation de connaissance seront étudiés. Enfin, les travaux actuels sur ces langages de « programmation » des années à venir seront décrits.

Les premières recherches en intelligence artificielle eurent souvent des buts ambitieux en s'attaquant à des domaines très vastes : traduction automatique sur des sujets quelconques, résolution générale de problèmes (GPS), démonstration de théorèmes dans toute théorie (Principe de Résolution). Or il est bien évident que chez l'homme, l'individu le plus doué dans son domaine intellectuel a des performances lamentables dans un autre. Ceci est dû à la masse considérable de connaissances que l'homme a su accumuler au cours des siècles dans toutes les disciplines.

Les systèmes d'intelligence artificielle ont notamment évolué ces dernières années afin de prendre en compte cette idée. Ainsi, il est aujourd'hui (1984) clair que, pour obtenir des programmes ayant des performances de très bonne qualité, il faut que ces programmes travaillent dans des domaines bien définis, sachent apprendre les connaissances dans cette spécialité, sachent les gérer et les structurer.

De tels programmes existent déjà et le sujet de ce chapitre est de faire le point sur ces **systèmes de représentation de grandes quantités de connaissances**. Leur

but est de parvenir dans des domaines précis à des performances égales à celles des meilleurs spécialistes. On les désigne sous le nom de « systèmes-experts ».

Un fait tout particulièrement important est que ces systèmes sont conçus pour l'homme, pour lui être effectivement utiles : il faut donc d'abord que ces systèmes puissent dialoguer en langage naturel avec le spécialiste.

Il faut ensuite que le système soit compris de ce spécialiste et qu'au besoin il explique et justifie son raisonnement aux yeux de celui-ci. Ceci implique que les connaissances déclaratives utilisées par le système soient bien séparées de leurs procédures d'utilisation et soient accessibles au système lui-même, que le raisonnement soit simple et robuste à la fois, pour être aisément explicable ensuite.

Il faut enfin et surtout que le spécialiste, que l'expert, puisse aisément transmettre ses connaissances. Il faut au besoin que plusieurs experts puissent enrichir chacun le système de leurs connaissances propres dans la spécialité. Ceci implique que cette connaissance puisse être entrée pêle-mêle, **en vrac, de façon purement énonciative** (sans préjuger de l'utilisation qui en sera faite, par opposition aux procédures figées classiques en algorithmique), et que le système lui-même organise, structure, compare cette connaissance avec ce qu'il sait déjà et mette enfin à jour son propre modèle de connaissance.

Nous aborderons l'étude des systèmes « experts » en présentant d'emblée deux dialogues réels d'un programme expert en médecine : MYCIN.

- . Un exemple de diagnostic.
- . Une explication de son propre raisonnement par le système.

En outre, une correction conversationnelle de la base des connaissances sera donnée au paragraphe **7.2**.

7.1 Exemples de dialogue avec un système « expert »

7.1.1 Un diagnostic médical

MYCIN : Mes recommandations thérapeutiques seront basées sur les infections probables et les organismes infectieux potentiels suivants :

L'organisme peut être : 1) pseudomonas aeruginosa ; 2) klebsellia pneumoniae ; 3) escherichia coli ; 4) bactéroïdes fragilis ; 5) enterobacter ; 6) proteus non mirabilis.

Voulez-vous me donner la gravité de la maladie de M. Fred Smith sur une échelle croissante de 0 à 4 ?

Utilisateur : 3

MYCIN : Monsieur Fred Smith présente-t-il des réactions allergiques à un quelconque agent microbien ?

Utilisateur : Non

MYCIN : Quel est son poids en kilos ?

Utilisateur : 70

MYCIN : Ma recommandation thérapeutique sera, en premier choix : pour couvrir les points < 123456 >, prescrire de la clindamycine, dose 595 mg 96 h, pendant 14 jours.

Désirez-vous éventuellement éliminer certains médicaments ?

Utilisateur : Oui, la clindamycine

MYCIN : Pour couvrir les points < 23456 >, donner du chloramphénicol, dose 1 g 96 h, pendant 14 jours.

** Surveiller le taux de globules blancs du patient.

Pour couvrir le point < 1 >, donner de la gentamicine, dose 119 mg 96 h, pendant 10 jours.

** Modifier la dose en cas de faiblesse rénale.

7.1.2 Explication, par le système, de son propre raisonnement

MYCIN : L'infection du malade a-t-elle été contractée à l'hôpital ?

Utilisateur : Pourquoi ?

TEIRESIAS : (Pourquoi est-ce important de savoir cela.)

Cela aidera à déterminer la catégorie à laquelle appartient l'organisme responsable de l'infection.

- Il a déjà été établi que : 1) L'organisme est à Gram négatif; 2) C'est un bâtonnet ; 3) Il se développe à l'air ou à l'abri de l'air.

- Donc, si l'infection par cet organisme n'a pas été contractée lors du séjour à l'hôpital,

- Alors il est presque certain (0,8) que l'organisme est de la catégorie des enterobacteriaceae ; de plus, il se peut ((0,1) ; règle37) que ce soit la pseudomonas aeruginosa.

Utilisateur : Pourquoi n'avez-vous pas envisagé la possibilité que l'organisme soit un streptocoque ?

TEIRESIAS : La règle 33 aurait dû être utilisée pour identifier l'organisme comme un streptocoque, mais la seconde clause « la morphologie de type coque » était déjà connue comme fausse et la règle 33 n'a jamais été essayée.

7.2 Le fonctionnement du système MYCIN-TEIRESIAS

MYCIN et l'ensemble des programmes qui le complètent - TEIRESIAS et BAOBAB - constituent un système-expert typique qui utilise de façon opérationnelle une importante masse de connaissances données sous forme énonciative (déclarative). Nous décrivons ici ses buts, son intérêt, son fonctionnement.

7.2.1 Le but

Le système MYCIN est conçu pour fournir de façon interactive une aide au médecin dans le diagnostic et le traitement des infections bactériennes du sang.

La consultation a lieu en anglais avec un médecin qui fournit toutes les informations utiles sur le patient et les résultats de laboratoire. Le raisonnement de MYCIN en absence d'informations complètes (les résultats des cultures ne sont communément connus qu'après 24 ou 48 h), est fondé sur un ensemble acquis de connaissances individualisées dites « règles de production » (cf. définition en 7.3.1) : MYCIN est capable d'expliquer son raisonnement et de répondre à des questions en anglais sur celui-ci. De plus, cette base de connaissances peut être changée ou augmentée et à tout moment, le système possède un modèle de sa connaissance et, en quelque sorte, sait ce qu'il sait.

7.2.2 L'intérêt

L'intérêt d'un tel système - l'un des plus achevés en intelligence artificielle - est double : les techniques mises en jeu sont générales et le programme, dans le domaine étudié, est opérationnel. Il obtient les résultats de qualité, effectivement utiles.

MYCIN, qui date du Ph. D. de Shortliffe (1974), est le maillon principal d'une chaîne de travaux menés à L'Université de Stanford dans l'équipe de John Mac Carthy au sein du Stanford Heuristic Programming Project qui date de plus de dix ans. Il a été mis au point en étroite collaboration avec les médecins du « Infection Disease Group » de l'Université.

Mais la même méthode et les mêmes chercheurs - Buchanan et Felgenbaum principalement - avaient déjà abouti à un système expert en chimie organique : « DENDRAL » et « METADENDRAL » (1968) et la même équipe pluridisciplinaire, intéressée aux différents aspects de l'automation du raisonnement scientifique a, depuis (1974), complété MYCIN par un système d'acquisition de connaissances - TEIRESIAS - écrit par R.Davis (1976) (cf. paragraphe 7.2.5) ainsi qu'un programme, BAOBAB, dû à Alain Bonnet (1980), de constitution automatique de la base de faits relative à un patient à partir de son dossier médical.

Puis cette équipe est passée à d'autres domaines, comme la réparation automobile, le traitement des méningites, le diagnostic des infections pulmonaires, l'aide à la prospection minière ou aux manipulations génétiques, ce qui montre la puissance et la généralité de l'approche adoptée. Par ailleurs, il convient de noter qu'un tel problème de diagnostic, même restreint à un domaine précis comme les infections microbiennes, est complexe. En effet :

- a) Il n'existe pas de panacée pour traiter les maladies infectieuses ;
- b) Des enquêtes révèlent qu'on donne par année de la pénicilline à une personne sur quatre, alors que cette prescription est inutile dans 9 cas sur 10 ; il faut donc vérifier la gravité de l'infection avant tout traitement ;
- c) Il faut donc aussi connaître l'organisme responsable de toute infection grave aussi bien que possible, mais ce problème est difficile : les analyses de laboratoires prennent 48 heures pour donner des caractéristiques assez précises, et plus pour faire un antibiogramme ; en outre, il faut se prémunir contre les agents infectieux qui vont probablement se manifester dans le futur proche ;
- d) Dans de nombreux cas, l'infection est tellement grave qu'il faut entreprendre un traitement avant tout résultat d'analyse et quelquefois - attaque bactérienne virulente après une opération cardiaque par exemple - en l'absence de tout spécialiste ;
- e) Pour compliquer encore, il existe des incompatibilités entre le patient et certains remèdes (allergies, problèmes de tension, d'âge...).

Tout système, homme ou programme, qui s'attaque à ce problème, doit avoir toutes les informations possibles sur tous ces points, savoir comment ils sont susceptibles d'interagir, et enfin être capable de prendre une décision dans un intervalle de temps très court. Cette décision n'est qu'un compromis effectué dans un univers imparfaitement connu entre l'efficacité présumée du traitement et le délai pour obtenir celui-ci.

En revanche, ce problème complexe est bien délimité : il n'est pas utile ici d'avoir des connaissances en médecine générale, pas plus qu'un modèle théorique de l'infection microbienne ou de la résistance des microbes aux antibiotiques.

Nous verrons que la représentation des connaissances sous forme déclarative, à l'aide de règles de production, se prête bien à ce type de domaine, qu'elle facilite en outre le dialogue en langue naturelle avec l'utilisateur, le programme étant en effet à même d'expliquer son raisonnement et qu'enfin un tel système constamment tenu à jour peut être un bon pédagogue, capable d'aider le spécialiste à actualiser son savoir.

7.2.3 Description

Le système se compose essentiellement d'une base de connaissances qui est un ensemble donné de règles et de quatre programmes principaux qui coopèrent suivant le schéma de la figure 7.1.

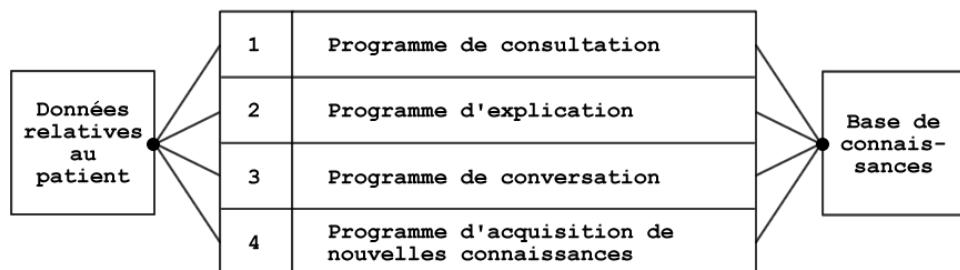


FIG. 7.1 – Schéma de principe de MYCIN

7.2.3.1 Les règles

La principale source de connaissances est une base d'environ 400 « règles de production » dont chacune possède un ensemble de prémisses décrivant une certaine situation et un ensemble d'actions à entreprendre si les prémisses sont toutes satisfaites.

Les règles sont la composante de base de ces systèmes. Chaque règle prend la forme :

Si P_1 et P_2 et... et P_i alors a_1 et a_2 ... et a_i .

Les P_i sont les prémisses ou conditions. Dans MYCIN, elles sont normalisées sous forme de quadruplets :

$<\text{prédicat}>$ $<\text{objet}>$ $<\text{attribut}>$ $<\text{valeur}>$.

Exemples :

MYCIN, règle 85 :

Si 1) Le site de la culture est le sang,
et si 2) L'organisme est à Gram négatif
et si 3) L'organisme est de forme bâtonnet,
et si 4) Le patient est un hôte à risque,
alors Il est probable (0,6) que l'organisme soit le pseudomonas aerugi-
nosa.

MYCIN, règle 217 :

Si 1) L'organisme est un bactéroïde,
et si 2) Le site de la culture était stérile,
alors la thérapie recommandée doit être choisie parmi les suivantes :
chloramphénicol, clindamycine, tétracycline, lindomycine, genta-
micine.

MYCIN possède un ensemble standard de 24 prédicats (par exemple : « est, le-même, connu, appartenant-à... »); 11 objets (« organisme, culture, médicament... ») et 80 attributs (« identité de l'organisme, sensibilité, site... ») qui servent à décrire le cas traité.

Les règles sont, au départ, fournies en langage naturel (cf. ci-dessus et [7.2.4](#)).

Il existe des règles pour le diagnostic et d'autres pour les prescriptions.

Chaque action est en outre affectée d'un coefficient d'atténuation, compris entre 0 et 1, qui exprime la plus ou moins grande certitude de l'expert qui a donné la règle vis-à-vis de la conclusion annoncée. La connaissance dans un tel domaine n'est en effet jamais absolue. Comme par ailleurs les faits observés - les symptômes - sont le plus souvent non-indépendants, les méthodes bayésiennes ne conviennent nullement. Nous allons voir que MYCIN travaille à partir de ces coefficients avec des facteurs de vraisemblance qui ne sont pas des probabilités.

7.2.3.2 Le raisonnement approché

Les faits concernant l'univers sont eux-mêmes représentés comme des triplets values par un coefficient de vraisemblance (C.V.) compris entre -1 et 1.

Exemples :

(Identité organisme : E. coll 0,7),
 (Identité organisme : Kiebsiella 0,4),
 (Sensibilité organisme : Pénicilline -0,9).

Le C.V. d'un nouveau fait, déduit d'une règle dont le C.V. propre est C.R., est donné par :

$$\text{C.V.} = \text{C.R.} \times \min CV_{PJ}$$

où CV_{PJ} désigne le C.V. de la j^{ime} prémissé.

Le modèle permet notamment la coexistence de plusieurs valeurs possibles différentes pour un même paramètre.

Si, lors du raisonnement, deux règles différentes donnent la même conclusion avec des C.V. $C1$ et $C2$, elles se renforcent mutuellement :

$$\text{C.V. conclusion} = C1 + C2 - C1 \times C2.$$

Tout fait dont le C.V. est inférieur en valeur absolue à 0,2 est considéré comme peu crédible et éliminé, par le système général, de la base des faits.

Certains auteurs critiquent ces calculs et proposent d'autres solutions (Friedman 1981, Doyle 1979, Swartout 1981).

7.2.3.3 Le mécanisme d'inférence

Le raisonnement lui-même se fait à rebours en partant des organismes - causes possibles de l'infection - vers les symptômes, par construction d'un arbre ET-OU.

Un but étant donné, le système considère toutes les règles dont les conclusions portent sur ce but. La partie de gauche (noeud ET) de chaque règle est, si possible, évaluée et donne alors un C.V. résultat V (valeur absolue) égal à celui de la prémissé de C.V. minimum :

si $|V| \geq 0,2$ la partie droite est formée et affectée d'un C.V. égal au produit de r par le coefficient d'atténuation de la règle.

si $|V| < 0,2$ la règle ne donne rien (l'hypothèse de départ n'est bien sûr pas niée).

Quand certaines prémisses ne peuvent être évaluées, elles sont empilées comme sous-buts intermédiaires (noeuds OU) et le système itère le processus (Voir infra le schéma du paragraphe 7.3.1).

Si le système ne parvient à aucune conclusion, il demande de nouvelles informations relatives au patient. C'est précisément pour ne pas poser trop tôt trop de questions au médecin (qui n'aime pas ça), tant qu'il n'a pas tout essayé (ce qui lui permet alors de choisir la bonne question), que MYCIN procède par une énumération exhaustive à rebours.

Cette énumération exhaustive sur des règles de production, qui s'apparentent à des règles de réécriture, rapproche MYCIN d'un démonstrateur de théorèmes travaillant dans un espace de recherche relativement réduit.

Quelques modifications améliorent la procédure standard :

MYCIN généralise les buts : « L'organisme est-il l'E. Coli ? » est systématiquement changé en : « Quelle est l'identité de l'organisme ? ». Si une telle recherche ne coûte

pas beaucoup plus cher au système, elle peut, par contre, éviter des questions à l'interlocuteur.

MYCIN utilise absolument toutes les règles, avec leurs coefficients d'atténuation pour chacun des buts, car dans ce domaine, on préfère peser et juger toutes les éventualités. Il n'y a que le cas où une inférence certaine (C.V. = 1 partout) est détectée que le processus est arrêté plus tôt.

MYCIN, pour être efficace, se donne malgré tout le droit d'interroger le médecin sur des attributs dont il n'a pas la valeur, mais dont il sait que ce sont des données de laboratoire courantes.

En outre, tout au long de cette recherche le système conserve la trace de toutes ses connaissances et l'historique en est mémorisé, ceci pour trois raisons :

1. Eviter des calculs inutiles ainsi que les bouclages par recherche d'informations qu'on ne peut obtenir ;
2. Etre capable d'expliquer à l'expert le comment et le pourquoi des conclusions auxquelles MYCIN parvient finalement (cf. paragraphe 7.4) ;
3. Savoir quelle question poser à l'expert dans la phase suivante, si des informations se révèlent faire défaut.

7.2.3.4 Les métarègles

Le système a accès à sa propre représentation de la connaissance et en a une certaine intelligence à travers : A) des profils ; B) des métarègles.

- A) **Les profils** sont attachés à chacun des 24 prédicats et permettent au système de savoir quels attributs commandent le prédicat en question. Ils permettent un gain de calcul en indiquant continuellement si tel attribut se retrouve de nombreuses fois dans différentes prémisses à l'intérieur de différentes règles. Ils autorisent donc une première évaluation rapide et grossière des prémisses. Cette évaluation indique, selon une logique simple, si une règle est applicable, en attente, ou bien inappropriée dans la situation courante.
- B) **Les métarègles** sont des données symboliques de structure analogue aux règles qui permettent au système de piloter sa recherche. **Elles expérimentent des stratégies** qui évitent l'énumération exhaustive en indiquant la meilleure approche pour un sous-but donné, les métarègles, **par des règles elles-mêmes**, d'où leur nom.

La première métarègle concerne le but général : « Proposer une thérapie pour un patient ».

« Si l'on recherche une thérapie,
alors, dans cet ordre, considérer les règles qui permettent de :

- 1) Acquérir les informations cliniques sur le patient ;
- 2) Trouver quels organismes, s'il en existe, sont cause de l'infection ;
- 3) Identifier les organismes les plus vraisemblables ;
- 4) Trouver tous les médicaments potentiellement utiles ;
- 5) Choisir les plus adaptés en plus petit nombre ».

Autres exemples de métarègles :

Métarègle 2 :

Si 1) le patient est un hôte à risque,
 et si 2) Il existe des règles qui mentionnent des pseudomonias dans une
 de leurs prémisses,
 et si 3) Il existe des règles qui mentionnent des klebsiellas dans une de
 de leurs prémisses,
 alors il est probable (0,4) qu'il faille utiliser les premières avant les
 secondes.

Métarègle 25 :

Si 1) le site de la culture est non stérile,
 et si 2) il existe des règles qui mentionnent dans leurs prémisses un orga-
 nisme déjà rencontré auparavant chez le patient et qui peut être
 le même que celui dont on recherche l'identité,
 alors il est sûr (1,0) qu'aucune d'entre elles ne peut servir.

[Une infection temporairement guérie peut réapparaître. Ainsi, il est habituel de rechercher la bactérie actuelle parmi celles qui ont déjà été rencontrées chez le même patient. Cependant cette méthode ne convient plus pour des cultures en milieu non stérile.]

La forme interne exacte de cette métarègle pour le programme est en LISP :

Prémisses : (\$AND (MEMBF SITE CONTEXT)
 (NONSTERILESITES)
 (THEREARE OBJRULES)
 (MENTION CONTEXT PREMISE SAMFBUG))
Action : (CONCLIST CONTEXT UTILITY YES TALLY-1)

Ainsi, les métarègles, qui ont le même format que les règles, sont balayées pour chaque sous-but. Elles peuvent réduire ou réarranger l'ensemble des règles, guidant l'analyse et coupant l'arbre de recherche. Comme chaque noeud de l'arbre a accès à toute l'information disponible, ces métarègles sont étroitement liées à ce noeud et la recherche s'adapte à chaque contexte.

C'est ainsi la structure de contrôle et les heuristiques qui sont mises en évidence dans les métarègles, au lieu d'être noyées dans le corps d'un programme.

Enfin, puisque l'interprète fait appel aux règles par leur contenu et non par leur nom, il permet au système de s'adapter naturellement aux additions, destructions ou modifications quelconques de la base de connaissances.

7.2.4 Explications fournies par MYCIN et dialogue en langue naturelle

Une exigence importante lors de la conception du système était que le programme sût expliquer pourquoi il faisait ce qu'il faisait, pour permettre aux médecins de comprendre à leur tour et donc de l'accepter! Des facilités de dialogue permettent ainsi à l'utilisateur d'interroger le système à tout moment.

La compréhension des questions et les explications sont effectuées par un interprète élémentaire grâce à des mots-clés et à des formats d'entrées-sorties tout faits. Le fonctionnement de cet interprète est grandement facilité par la forme standard de la connaissance (les règles de production) et par le type très simple de la recherche (combinatoire).

Comme pour le programme TEIRESIAS dont l'exposé suit, nous ne détaillerons pas plus avant cette partie concernant le langage naturel qui n'est pas ici notre propos, et n'est de toute façon pas l'apport principal de ce travail.

7.2.5 L'acquisition de nouvelles connaissances : TEIRESIAS

TEIRESIAS est un programme interactif destiné à assister un expert dans l'élaboration de systèmes fonctionnant sur une grande masse de connaissances. C'est une interface dialoguant avec l'expert en langue naturelle pour lui éviter de descendre au bas niveau de l'implémentation. Le système peut non seulement utiliser sa connaissance directe mais aussi l'examiner, l'abstraire, raisonner sur elle, (Teiresias est, dans l'Oedipe-Roi de Sophocle, le nom de l'oracle aveugle). Le dialogue peut ainsi s'instaurer de spécialiste à spécialiste.

A l'origine TEIRESIAS étend le système MYCIN aux infections quelconques du sang (Ph. D. de R. Davis, Stanford 76) mais son but est, plus précisément, en aidant à la constitution de la base de connaissances, de pouvoir facilement passer à un autre domaine, ce qui a plusieurs fois été fait depuis (cf.paragraphe 7.4).

Le seul présupposé important est que la représentation (règles de production) et les concepts sont bons et que l'on cherche seulement à augmenter ou à modifier la base des données. Tout se passe comme si l'expert-professeur avait enseigné un cours, donnait ensuite de nouveaux problèmes à l'élève, regardait avec attention les performances de ce dernier et les corrigeait.

7.2.5.1 Exemple commenté de TEIRESIAS au travail

Le dialogue donné à la figure 7.2 sert de base à la suite de l'exposé. Il a été obtenu par Davis, pour être plus lisible, en remplaçant **manuellement** le vocabulaire médical par un vocabulaire boursier (E. Coli —> American Telephones & Telegraph, infection donne investissement, prendre un médicament donne acheter un titre, etc.).

TEIRESIAS sait tenir compte, de façon interactive, de l'avis de l'expert après toute consultation :

TEIRESIAS : Les investissements suivants semblent être actuellement les plus appropriés : Varian Corporation et American Telephone and Telegraph (AT & T). Est-ce correct ? (1)

Expert : Non.

TEIRESIAS : Y en a-t-il qui manquent ?

Expert : Oui, Digital Equipment et Data General.

TEIRESIAS : Y en a-t-il qui ne devraient pas apparaître ?

Expert : Oui :AT & T.

TEIRESIAS : Voulez-vous que je guide la recherche de l'erreur ?

Expert : Oui.

TEIRESIAS : Commençons alors par AT & T. (2)

J'ai utilisé la règle 27 pour y parvenir, puisque :

- 1) Le domaine de l'investissement n'est pas précisé ;
 - 2) Le bénéfice attendu est supérieur à 10 % ;
 - 3) L'investissement doit se faire à long terme ;
- Alors AT & T est un choix plausible (0, 4).

La règle est-elle correcte ?

Expert : Oui.

TEIRESIAS : Une prémissse n'aurait-elle pas dû être vérifiée ?

Expert : Oui, la 1 (3)

TEIRESIAS : Je ne suis parvenu à aucune conclusion au sujet du domaine de l'investissement. Aurais-je dû trouver quelque chose de précis ?

Expert : Oui. La technologie avancée.

TEIRESIAS : Les essais pour toutes les règles qui pouvaient conclure au domaine de la technologie avancée ont tous échoué car les prémisses marquées étaient fausses : (Impression des deux règles en cause).

Ces deux règles étaient-elles correctes ?

Expert : Oui.

TEIRESIAS : L'une d'entre elles aurait-elle dû s'appliquer ?

Expert : Non. (4)

TEIRESIAS : Manque-t-il une règle ?

Expert : Oui.

TEIRESIAS : OK, alors j'ai besoin d'une règle qui me permette de déduire que le domaine doit être la technologie avancée.

Voulez-vous me donner une telle règle ?

Expert : Si la tranche d'imposition du client est 80%,

si le client suit la tendance boursière de près,

(The client is following up on market trends carefully),

alors il est évident (0, 8) que l'investissement doit se faire dans la haute technologie. (5)

TEIRESIAS : La nouvelle règle est appelée règle 383.

Voici comment je la comprends. Si :

- 1) La tranche d'impôt du client est 50%,
- 2) Le marché a été à la hausse récemment,
(the market has followed an upward trend recently)
- 3) Le client suit son portefeuille de près,
(the client manages his assets carefully)

Alors l'investissement doit se faire en haute technologie (0, 8).

D'accord ?

Expert : Non (6)

TEIRESIAS : Il faut changer les prémisses ?

Expert : Oui. changez 2).

TEIRESIAS : Supposez que 2) soit changée en « 2) le client suit le marché attentivement », est-ce que cela irait ?

Expert : Oui.

(7)

TEIRESIAS : Sans doute voulez-vous alors supprimer 3).

D'accord ?

Expert : Oui.

(8)

TEIRESIAS : (Répétition de R383, puis) :

Je n'aime pas critiquer, Randy, mais savez-vous que généralement, les règles au sujet des domaines d'investissement qui mentionnent la tranche d'imposition et la façon de suivre le marché du client mentionnent aussi l'expérience du client en matière d'investissement ? Dois-je tenter de formuler une troisième prémissse ?

Expert : Oui.

(9)

TEIRESIAS : Que pensez-vous de : le client a en matière d'investissement une expérience modérée ?

Expert : D'accord.

(10)

TEIRESIAS : La plupart des mêmes règles donnent aussi une conclusion sur la classe de risque désirée pour l'investissement. Dois-je tenter de donner une conclusion semblable ici ?

Expert : Non.

TEIRESIAS : (Répétition de R383 corrigée, puis) :

S'il vous plaît, dites brièvement pourquoi vous avez ajouté cette règle.

Expert : Le système n'avait pas compris qu'il pouvait préciser le domaine de l'investissement, ce qui a permis à la règle 27 d'être appliquée à tort.

TEIRESIAS : R383 a été ajoutée à la base de connaissances.

La consultation va être relancée pour tester effectivement le bien-fondé de la règle.

Installez-vous confortablement, cela peut prendre un peu de temps...

Résultats : investissements proposés : Varian Corporation, Data General, Digital

Equipement.

Bien, il apparaît que tout est corrigé n'est-ce-pas ?

Expert : Oui.

FIG. 7.2 – Un dialogue avec TEIRESIAS

Les numéros entre parenthèses renvoient aux commentaires ci-après.

- 1) C'est ici, après les résultats fournis par MYCIN, que TEIRESIAS entre en scène. La première phase va consister à dépister l'erreur.
- 2) Des heuristiques tirées de l'expérience ont été données au programme sous forme de métarègles. L'une d'elles dit qu'il convient :
 - a) de traiter un problème à la fois,
 - b) de commencer de préférence par une conclusion qui n'aurait pas dû être obtenue.
- 3) L'expert indique que cette règle n'aurait pas dû s'appliquer. Dans le cas contraire, TEIRESIAS aurait signalé qu'une règle manquait sans doute, qui interdirait

AT & T.

- 4) Cela aurait été le cas s'il y avait eu erreur sur la valeur d'une prémissse donnée pour fausse par une autre règle.
- 5) Le dépistage de l'erreur, dirigé par les questions précises et pertinentes du système a été remarquablement rapide.
Dans cette deuxième phase le système va interpréter la nouvelle règle.
- 6) L'erreur de TEIRESIAS est due à une analyse linguistique mot à mot très rudimentaire (pas même d'analyse syntaxique) : le prédicat « is » est associé à l'objet « market ». Le système pense d'abord à une prémissse du type (la tendance du marché est..) qui est une phrase modèle connue du système (cf. [7.3.2](#)) renforcée ici par le « up » qui suit, donnant : « à la hausse ».
- 7) L'erreur est ensuite facilement corrigée. Le « deuxième choix » est correct. La clause 3 est détruite car on ne peut plus comme précédemment supposer que la phrase se découpaît en deux prémisses indépendantes puisque le système sait notamment que « carefully » ne peut être utilisé deux fois. Cela marche car ce sont de petits textes, que peu d'analyses différentes sont possibles, et enfin que des tests de correction a posteriori sont facilement effectués.
- 8) Voilà achevé le premier test de compréhension :
 - Lecture du texte d'origine en anglais
 - Mise sous forme de représentation interne
 - Retraduction en anglais, demande d'approbation.

A la fin de cette phase l'expert est satisfait ; dans la dernière phase, **le système va voir s'il est, lui, satisfait.**

- 9) Voir le paragraphe suivant qui explique le mécanisme de création de la nouvelle règle par le système.
- 10) La règle doit en tout cas s'appliquer à la situation courante : « expérience modérée » (cf. le programme de Waterman au poker, paragraphe [7.3.2](#)).

7.2.5.2 Fonctionnement

Le système demande à l'expert de juger son raisonnement. Cela n'est possible que parce que :

- 1) Le raisonnement est fondé sur une méthode de recherche rudimentaire à partir de règles externes simples et non sur un programme de type habituel.
- 2) Le système a gardé en mémoire les effets de ses actions et sait voir quelles sont celles qui ont empêché la bonne conclusion. Il est donc à même de formuler toutes les hypothèses possibles de correction ainsi que leurs effets attendus.
- 3) Le système, loin de poser des questions vagues et générales, demande : « Dans cette situation précise, qu'est-ce que vous savez et que je ne sais pas ? »

TEIRESIAS a effectivement une connaissance a priori de ce qu'il sait et aussi, sous réserve qu'il y ait régularité dans la connaissance, de ce qu'il va rencontrer, par l'intermédiaire des modèles de règles.

- **Les modèles de règles**

Ces modèles sont des descriptions succinctes de haut niveau, décrivant, par abstraction et généralisation, des sous-ensembles de règles du niveau inférieur qui ont quelque chose en commun. Un modèle de règles se compose de quatre parties :

- Une liste d'exemples : c'est simplement le sous-ensemble des règles à partir desquelles le modèle a été constitué ;
- Une description d'un élément caractéristique du sous-ensemble, ici : caractérisation des prémisses et des conclusions par les attributs qui apparaissent fréquemment et corrélations de ces attributs entre eux : ces renseignements sont obtenus par une analyse statistique sommaire ;
- Deux listes de règles les unes plus spécifiques, les autres plus générales. L'ensemble des modèles est ainsi organisé en un arbre dont la racine est le concept le plus général.

C'est TEIRESIAS lui-même qui fabrique ces modèles à mesure que croît sa base de connaissances et dès que plusieurs règles portent sur un même domaine.

- *Exemple de modèle de règle :*

Modèle « *x est-un-domaine-d'investissement* » :

Exemples : R116, R80, R95...

Description :

- prémisses : le revenu est/la durée est/la tendance est/
 - corrélations : (revenu, durée)/(tranche impôts, expérience, attention)/
- Conclusions : domaine/risque.

Plus général : « *un-domaine-d'investissement* ».

Moins général : « *entreprises-de-services-publics-est-un-domaine-d'investissement* ».

Quand TEIRESIAS s'attend à recevoir une nouvelle règle (phase 3 dans l'exemple), il parcourt l'arbre des modèles en partant de la racine et en cherchant à retrouver ce qu'il sait déjà à propos de la règle. Ici, le modèle le plus spécifique connu qui lui permet de conclure qu'un domaine d'investissement possible est la technologie avancée, est précisément celui donné dans le modèle ci-dessus.

C'est la fréquence d'apparition des prédicats dans les modèles qui facilite la compréhension des clauses entrées en langue naturelle, tandis que les corrélations entre occurrences de ces mêmes prédicats permettent de vérifier que l'expert n'a rien oublié d'important.

C'est d'ailleurs cette utilisation des modèles qui supplée à la faiblesse de l'analyse linguistique : c'est elle aussi qui fait qu'une caractéristique importante du système est de tendre à ne comprendre que ce qu'il s'attend à comprendre.

L'arbre des modèles de règles permet en outre à TEIRESIAS de mesurer ce qu'il ne sait pas : domaines où existent peu de règles ou peu de règles sûres. Il sait ainsi ce qui est pour lui le plus urgent d'apprendre et peut donc poser lui-même des questions sur ces sujets (par contre il n'a pas de mesure de ce qui reste en fait à apprendre et n'a aucun moyen de juger d'entrée de jeu la vraisemblance d'un fait nouveau pour lui). Le même arbre permet enfin une mise en forme de la connaissance donnée en vrac par l'expert et de la structurer au profit des experts eux-mêmes.

7.2.6 Résultats et Analyse critique

Le système MYCIN-TEIRESIAS est écrit en LISP sur PDP 10. Cinq spécialistes, n'appartenant pas à l'équipe, ont jugé MYCIN sur 15 cas : valeur des thérapies conseillées, opportunité des questions posées, questions importantes oubliées. Dans 72% des cas ils ont approuvé le programme ; **dans la majorité des cas restants, ils n'étaient pas d'accord entre eux.**

Les nombreux systèmes experts, dont une revue est faite au paragraphe 7.3, montrent à l'évidence l'intérêt d'un programme comme TEIRESIAS. Le système PUFF pour l'aide au diagnostic dans les maladies pulmonaires a pu ainsi être achevé après moins de 50 heures de travail en collaboration avec les experts et moins de 10 semaines-homme de mise au point.

Il est indiscutable qu'au moins dans certains domaines les systèmes avec règles et métarègles procurent une solution tout à fait satisfaisante au problème de la transmission et l'utilisation de grandes quantités de connaissance.

Les avantages, les inconvénients, les caractéristiques et les présupposés de tels systèmes seront discutés de façon globale au paragraphe 7.7. On notera les limitations suivantes du système MYCIN-TEIRESIAS :

Le traitement du langage naturel est la partie faible ; il a été amélioré par A. Bonnet (Bonnet 80) avec le programme BAOBAB tant pour la compréhension des règles que pour l'entrée du dossier médical du malade en utilisant au maximum la connaissance a priori du sujet grâce à des frames analogues à ceux de Bobrow et Winograd (Bobrow 75) (cf. paragraphe 7.6), mais tous les problèmes ne sont pas résolus.

Si MYCIN a échoué parce qu'il faut modifier ou introduire simultanément plusieurs règles liées les unes aux autres, TEIRESIAS échouera à sa suite car le cas n'est pas actuellement prévu.

Lors de l'introduction d'une nouvelle règle, TEIRESIAS suppose a priori la règle douceuse et la banque de connaissances correcte : le contraire peut se produire et il y aurait alors quelques problèmes.

Mais le système dans son ensemble est bel et bien opérationnel et utile, il gère effectivement une connaissance spécialisée, entrée en vrac. Il a un modèle de cette connaissance qu'il crée et met à jour lui-même et qui lui permet de dialoguer en expert avec l'expert et d'améliorer continuellement par l'expérience cette connaissance.

C'est au total un système très simple dans sa définition et sa réalisation, un système dont le comportement est aisément compris, vérifié et admis par l'expert, un système intelligent autonome, parmi les mieux conçus.

7.3 Les systèmes de production

Ce paragraphe a d'une part pour but de définir plus précisément ce qu'est un système-expert et comment il est possible de le programmer, et d'autre part de faire le bilan des systèmes-experts actuels.

Une règle de production est une expression de la forme :

$$MG \rightarrow MD$$

dans laquelle le membre gauche MG décrit une certaine situation, représentée dans un formalisme approprié pour l'univers de travail, et où le membre droit MD donne l'action **à envisager** lorsque la situation associée est détectée. Une telle règle ressemble fort à un théorème et un enchaînement de règles emboîtées par leurs prémisses MG et leurs conclusions MD s'apparente à un syllogisme. Le mode de raisonnement sous-jacent des systèmes qui utilisent comme base de connaissances des ensembles de telles règles est le modus ponens (de p et $p \supset q$ on conclut q). Il est à ce titre aisément compréhensible lors de l'analyse des résultats, mais ce n'est pas le seul (cf. paragraphe 7.4).

Le schéma de calcul par règles de production, s'il s'oppose en pratique aux systèmes procéduraux (paragraphe 7.5), ne leur en est pas moins formellement équivalent. Il fut proposé la première fois par E. Post (Post 36) et notamment utilisé depuis pour définir les langages et les grammaires formelles (Chomsky 63). Il est à la base, depuis 1974, de nombreux programmes, qualifiés de « systèmes experts ». Ces systèmes sont spécialisés dans des domaines particuliers où la résolution de problèmes demande des raisonnements simples tout en présentant un fort aspect cognitif. Ce sont des règles de production qui sont chargées de supporter toute la connaissance. Cette connaissance est ainsi donnée de façon modulaire et aisément modifiable, puisqu'elle n'est pas mêlée au corps des programmes. La figure 7.3 donne un exemple de dérivation dans un tel système.

(a)	(b) : H,K	
(R1) $A \rightarrow E$		
(R2) $B \rightarrow D$	$H \rightarrow A$	(R3)
(R3) $H \rightarrow A$	$A \rightarrow E$	(R1)
(R4) $E \wedge G \rightarrow C$	$E \wedge K \rightarrow B$	(R5)
(R5) $E \wedge K \rightarrow B$	$B \rightarrow D$	(R2)
(R6) $D \wedge E \wedge K \rightarrow C$		$D \wedge E \wedge K \rightarrow C$ (R6)
(R7) $G \wedge K \wedge F \rightarrow A$		
(a)	(c)	

(a) Règles, (b) Base de faits initiale, (c) Chaîne de dérivation

FIG. 7.3 – Exemple de dérivation en chaînage avant

Les prémisses A, B... peuvent être de simples propositions logiques « vraies ou fausses » - comme dans MYCIN - ou des prédicats avec variables ou encore des relations quelconques. La richesse du langage et la facilité d'expression des connaissances augmentent, bien sûr, avec la puissance des formalismes utilisés (cf. paragraphe 7.4 et suivants).

Par ailleurs, il est remarquable que les mêmes systèmes de production servent actuellement et concurremment à des psychologues pour tenter de modéliser certains processus intellectuels. A. Newell pense par exemple (Newell 72) que c'est le bon formalisme pour exprimer les transferts entre mémoire à court terme et mémoire à long terme chez l'homme et que ces systèmes constituent un modèle correct de la gestion de nos connaissances.

Leur utilisation en I.A. n'est donc pas une simple coïncidence !

7.3.1 Description d'un système de production

Le système global, outre sa banque de connaissance qui est l'ensemble des règles de production, se compose d'un espace de travail (ou base des faits) et d'un interprète programmé.

L'espace de travail contient l'ensemble des faits que le programme a pu déduire à un instant donné. Seul y figure initialement l'énoncé du problème à résoudre. Cet espace joue le rôle d'une « mémoire à court terme » qui contient de simples assertions relatives à la description statique de l'univers et est ainsi fonctionnellement différent de la banque de connaissances dynamiques de la « mémoire à long terme » qui contient les opérateurs des transformations sous forme de règles. Ces règles peuvent contenir des variables dont les valeurs sont précisées (« instantiées ») par l'interprète lors de chaque exécution pour les mettre en accord avec les faits connus par « semi-unification » ou « filtrage ».

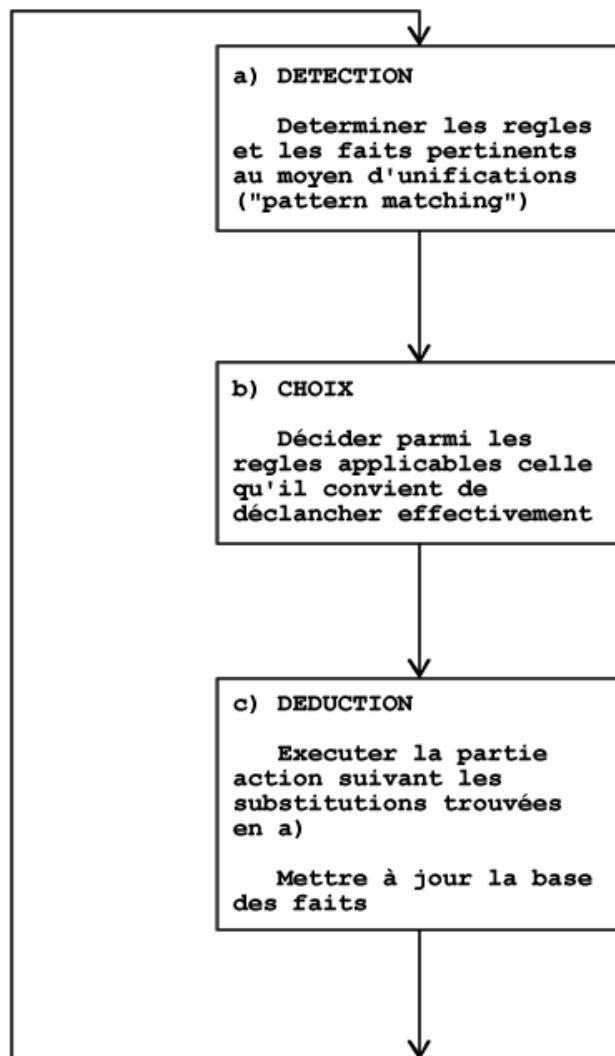


FIG. 7.4 – Cycle de base d'un interprète de règles

L'unification est ainsi le mécanisme fondamental qui permet aux systèmes de produc-

tion de fonctionner : dans toute sa généralité il permet de dire s'il existe un jeu de substitutions de variables qui permet de rendre deux formules logiques identiques. Il lui correspond un algorithme parfaitement défini (cf. le chapitre 3 : Systèmes formels).

L'interprète est alors une partie clé du système qui contrôle complètement l'ordre des déductions successives. Une exécution a toujours lieu suivant une séquence de cycles élémentaires parfaitement semblables les uns aux autres, dont la forme est donnée par la figure 7.4.

Le problème crucial est le problème du « déclenchement » : Plusieurs solutions ont été retenues, que nous étudierons en 3.3.

Les systèmes de production diffèrent ici selon que l'information nouvelle obtenue par cette déduction vient s'ajouter simplement à la base des faits ou bien est ajoutée en détruisant et en remplaçant l'un des faits qui y étaient présents ; voir par exemple (Post 36) ou (Newell 71).

La recherche du but elle-même se fait de façon naturelle, comme en démonstration de théorèmes, et est aisément représentable à l'aide d'un arbre ET-OU (fig.7.5).

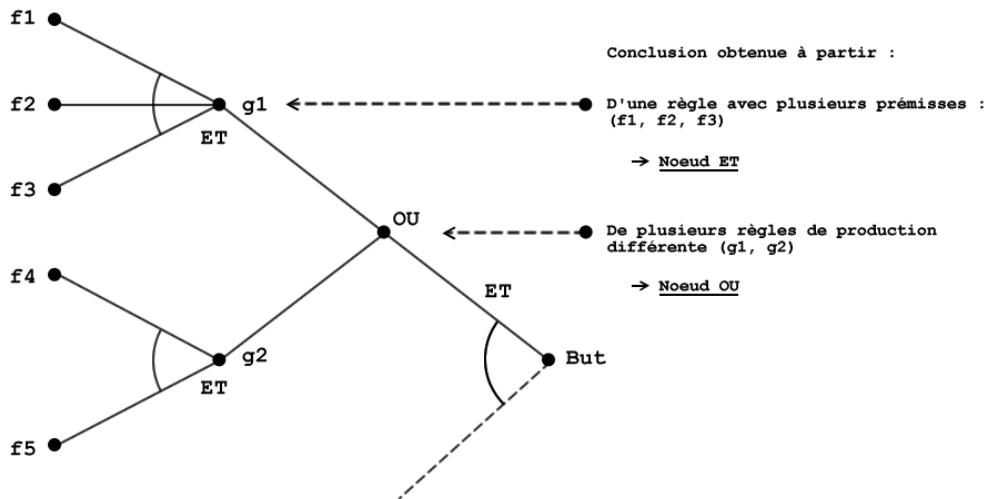


FIG. 7.5 – Arbre de déduction en chaînage arrière avec les mêmes données qu'à la figure 7.3

En fait, cet arbre potentiel peut être construit et parcouru soit depuis les faits vers la conclusion, soit depuis la conclusion, considérée alors comme conjecture, vers les faits. On parle dans le premier cas de *chaînage avant* (fig 7.3), dans le second de *chaînage arrière* (fig 7.5). Le plus souvent, c'est cette dernière solution qui est choisie, car d'une part les faits ne sont pas indépendants et d'autre part beaucoup d'entre eux ne sont pas pertinents pour le problème ; cette démarche évite ainsi, de prime abord, l'explosion combinatoire car peu de conclusions sont a priori envisageables. Elle est adoptée notamment dans MYCIN et DENDRAL (cf. 7.3.4).

Un interprète LISP, par appels récursifs en cascade, assure dans ce cas, la majeure partie du travail d'évaluation. Le schéma général d'un tel système est donné par la figure 7.6. Mais notons qu'un bon interprète doit en réalité, pour des raisons naturelles de commodité d'emploi, être capable, suivant l'étape et le type du raisonnement, de cheminer **en avant ou en arrière**.

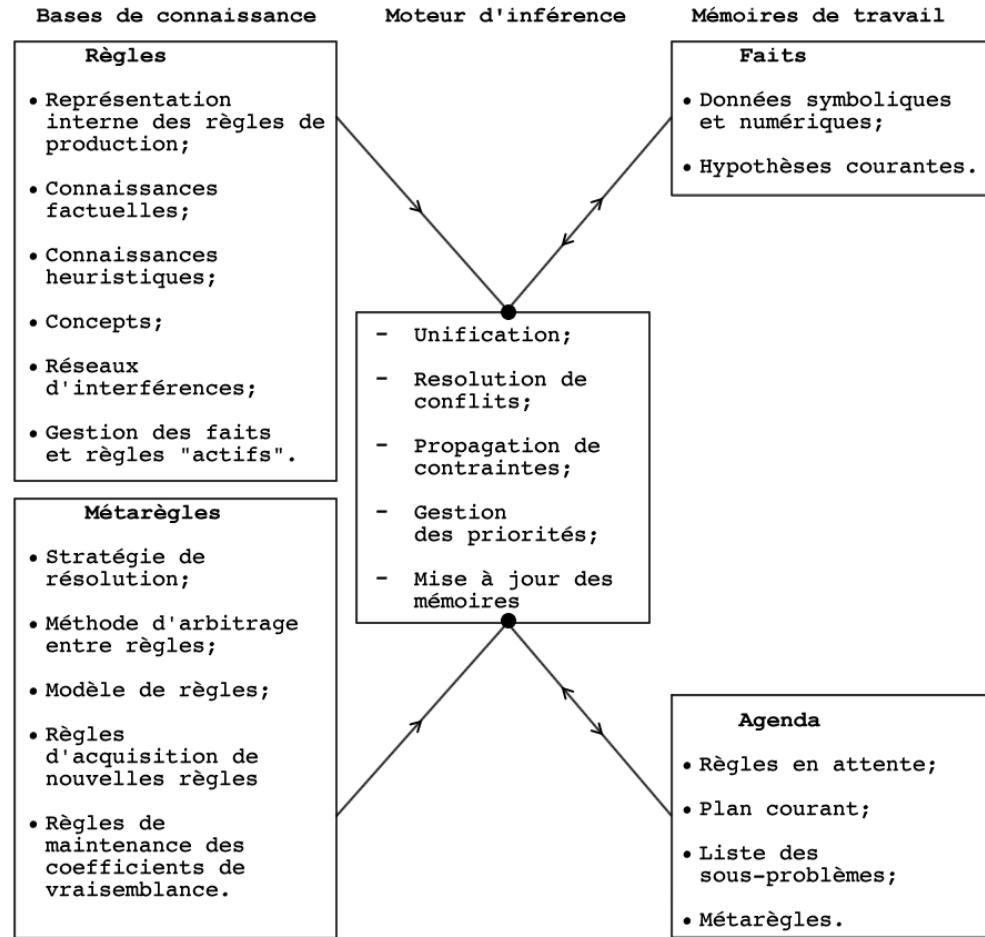


FIG. 7.6 – Schéma général d'un système actuel de règles de production

7.3.2 Hypothèses et domaines d'applicabilité des systèmes de production

La façon de faire entrer et d'utiliser la connaissance qui vient d'être décrite recèle plusieurs présupposés.

7.3.2.1 Rôle du spécialiste

Il faut tout d'abord qu'un spécialiste du domaine soit effectivement capable de décrire la situation. La science en question doit être suffisamment avancée pour que de bons concepts aient déjà été élaborés. Dans le cas contraire, il faudrait descendre à un tel niveau de détail que la taille de la base de connaissances à constituer deviendrait rédhibitoire (Nilsson 80).

7.3.2.2 Structure de l'univers étudié

Il faut énoncer les règles. Cela implique une interaction faible entre les paramètres qui décrivent une situation sous peine d'avoir à spécifier, dans le cas contraire, des membres gauche démesurément longs et inextricables. L'univers étudié doit donc être faiblement lié : chaque action décrit alors à l'intérieur de celui-ci une connaissance élémentaire mettant en jeu un petit nombre de paramètres.

En revanche, on remarque qu'une seule règle peut malgré tout résumer une quantité importante de savoir. Ainsi, « un organisme du sang Gram négatif et de type bâtonnet est vraisemblablement un E. Coli » est une règle basée à la fois sur des connaissances en physiologie et sur l'expérience clinique.

7.3.2.3 Schémas d'inférence

Le domaine doit être justifiable du schéma d'inférence simple, du type modus ponens (p et $p \supset q$ donne q) ou de sa variante modus tollens ($\neg q$ et $p \supset q$ donne $\neg p$).

Le but assigné doit notamment être bien défini et précis (d'où possibilité de cheminement arrière), les variables de la recherche varient de façon discrète plutôt que continue pour éviter une explosion de la base des faits. Les tâches à résoudre ne doivent pas faire appel à de nombreux sous-but reliés fortement et séquentiellement les uns aux autres, sinon l'approche procédurale classique s'impose.

7.3.2.4 Conséquences a priori de ces hypothèses

Ce type de modèle ne convient pas pour résoudre des problèmes complexes dans leur énoncé ou dans leur solution (problèmes trop combinatoires par exemple).

Le modèle est, en outre, par nature, polarisé plus sur les modifications du domaine que sur les propriétés d'invariance. Les tautologies, définitions ou relations de dépendance formelle ne semblent pas relever en effet du même mécanisme de base. Elles peuvent être prises en compte par d'autres schémas d'inférence : « démons » (Hewitt 72), « interruptions prioritaires » (Gascuel 81), « classes de similitude » (Bundy 79) (cf. paragraphe 7.4).

7.3.3 Variantes utilisées dans la conception des systèmes de production

Les deux points importants, lorsqu'il s'agit d'utiliser l'information à travers un tel système, concernent la représentation de la connaissance et la structure de contrôle.

7.3.3.1 Le contenu de la base d'expertise

La nature véritable des entités présentes est susceptible de varier d'un système à l'autre pour s'adapter au domaine. Le plus souvent, un format fixe est adopté pour les règles. Les paramètres sont typés pour restreindre les unifications autorisées.

Ainsi, dans DENDRAL, le membre gauche est le nom d'un squelette (graphe d'une partie de molécule) et le membre droit décrit les transformations possibles de ce graphe dans le spectographe. Une règle s'écrit alors par exemple :

« Estrogène → casse ((14 – 15), (13 – 17)) et
transfert hydrogène (1, 12) »

Concision et efficacité découlent de cette écriture conventionnelle compacte.

Enfin, tandis que quelquefois les prémisses et les conclusions des règles sont des faits de base qu'il suffit de tester ou de modifier, plusieurs systèmes admettent aujourd'hui dans les règles des procédures lancées par la règle appelante suivie d'une évaluation classique.

7.3.3.2 La structure de contrôle

Le problème des structures de contrôle concerne « la résolution du conflit » entre toutes les règles candidates. La stratégie de choix est bien sûr vitale, non seulement pour les performances, mais également et surtout pour la capacité du système à comprendre ce qu'il fait et s'améliorer.

Trois grandes familles de structures de contrôle se rencontrent actuellement dans la littérature :

- La recherche exhaustive,
- Le choix par évaluation,
- Le contrôle par métarègles.

Recherche exhaustive

C'est notamment la recherche de MYCIN, décrit en détail au paragraphe 7.2 (qui en outre travaille à rebours de la conclusion vers les faits). Ce type de contrôle s'adapte bien chaque fois que la réponse à la question précise posée passe par une pondération du jugement liée à toutes les conclusions possibles.

Ce type de contrôle suffit lorsque l'univers de recherche est combinatoirement petit.

Choix par évaluation

L'évaluation la plus sommaire est de s'arrêter à la première règle rencontrée qui convient (voir par exemple Waterman ou Moran en psychologie). Cette façon de procéder, dont l'efficacité est donc liée au rangement des règles au départ, réduit à néant la plus fondamentale des qualités des systèmes de production qui est de pouvoir sans problème modifier la base de connaissances.

Dans les autres cas, on collecte d'abord toutes les règles candidates puis on choisit celle de plus forte évaluation selon un critère ou un autre, suivant les systèmes :

- I) Ordre par rapport à une priorité des règles évaluée dynamiquement en fonction de l'intérêt de leur partie conclusion par rapport au but ;

-
- II) Ordre par rapport aux faits : priorité à la règle qui s'unifie avec le fait jugé le plus important ;
 - III) Critère de spécialisation : choix de la règle qui est la plus spécifique (critère de l'interprète LISPL) ;
 - IV) Choix en fonction d'un graphe traduisant un ordre partiel sur les règles (DEN-DRAL) ou construit par programme (Gascuel 81) ;
 - V) Choix de la règle la plus récemment utilisée ;
 - VI) Choix de la règle qui mentionne le fait le plus récemment utilisé.

Contrôle par métarègles

C'est de loin l'attitude préférable à toutes les autres, puisque d'implicite à l'intérieur de l'interprète, la logique de raisonnement devient explicite dans une forme semblable aux règles elles-mêmes. La démarche du programme est à chaque instant très dépendante de l'état de la recherche. Les métarègles sont là pour dire ce qu'il faut essayer en priorité dans chaque situation particulière. L'idée est la même que les conseils donnés à Planner avec les ordres « THUSE » pour démontrer un théorème (Hewitt 72).

Notons que, dans tous les cas, la structure de contrôle, par sa nature même, impose au système que toute communication se fasse à travers la base de faits. L'esprit même des règles de production interdit aux règles de s'appeler l'une l'autre (exactement à l'inverse de ce qui se passe couramment pour les procédures en programmation classique). (Voir le paragraphe 7.5 pour une comparaison plus fine). La propriété importante liée alors aux métarègles est que l'on peut modifier, ajouter, supprimer des règles élémentaires et que le système s'en arrange tout seul.

7.3.3.3 Coût d'exécution d'un système de production et efficacité

En première approximation, le temps de calcul et de construction d'un raisonnement croît comme le produit du nombre de règles et du nombre d'enregistrements dans la base de faits. Le facteur le plus important est le nombre d'essais infructueux pour déclencher les règles. Différents filtres peuvent être construits qui éliminent, avant l'unification proprement dite, les règles et les faits inutiles, J. Mc Dermott (Dermott 78) en propose quatre et exhibe des formules correspondantes d'évaluation des temps. Une solution retenue par plusieurs auteurs consiste à propager, après un déclenchement donné, les faits déduits en ne « réactivant » que les seules règles qui mentionnent ces faits comme prémisses. En outre, certains interprètes sont conçus pour n'évaluer qu'une fois par cycle les sous-expressions communes à plusieurs règles ou bien même les sauvegarder d'un cycle à l'autre, tant que les faits sur lesquels elles portent ne sont pas modifiés (Gascuel 81), (Ghallab 81) (Forgy 81). Des heuristiques simples et peu coûteuses jouent ici à plein rendement : en très peu de temps elles diminuent l'effort nécessaire pour apparier les règles et les faits.

L'agrégation automatique des faits élémentaires en structures plus complexes ou représentations de haut niveau est un autre moyen de réduire les essais : la probabilité de réussir l'unification diminue avec la richesse des objets considérés et le nombre d'essais inutiles diminue parallèlement. La structure de prototypes (*frames*) a ainsi été proposée par Minsky (75) et est utilisée dans plusieurs systèmes (cf. paragraphe 7.5).

Ainsi le problème est relativement simple quand il n'y a pas de variables dans les règles. Quand au contraire les variables sont autorisées et lorsqu'en outre le nombre de règles

(plusieurs centaines), ainsi que le nombre de faits (plusieurs milliers) sont grands, le problème reste entier et l'explosion combinatoire ne semble pouvoir réellement être maîtrisée qu'à l'aide de métacognition (cf. paragraphe 7.7).

Nous passons maintenant en revue les systèmes experts actuels.

7.3.4 Différents systèmes experts utilisant des règles de production

7.3.4.1 DENDRAL

Sujet : Donner la formule développée d'un corps organique à partir de sa formule brute et de son spectogramme de masse.

Origine : Université de Stanford, Buchanan, Felgenbaum, Lederberg (prix Nobel de Médecine 1958), Sutherland & Co avec le Massachussets Spectrum Laboratory.

DENDRAL est issu des travaux initiaux de Lederberg qui avait, dès 1964, mis au point un système d'aide à l'analyse en chimie organique (construction et codage des molécules en tenant compte des contraintes topologiques). La croyance, à cette date, dans les capacités d'un système expert est étonnante car la mode est alors aux programmes généraux, capables de tout faire à partir de la démonstration de théorèmes et du principe de Résolution (Felgenbaum 71).

A l'origine DENDRAL était un programme procédural de type habituel (Davis 77). Il fut complètement réécrit en 1967 car il devenait ingérable : la nécessité d'entrer en données le savoir des spécialistes sauta alors, pour la première fois, aux yeux des programmeurs.

Caractéristiques : La méthode utilisée est de type recherche énumérative. Un programme combinatoire engendre toutes les structures possibles satisfaisant les contraintes liées aux données. Il est très efficace et beaucoup plus rapide qu'un chimiste. Une formule plausible étant obtenue, les règles de la forme : « telle configuration atomique → telle fragmentation » permettent ensuite de prédire son spectre et de le comparer avec l'observation réelle.

D'une manière générale et, soit parce que plusieurs règles peuvent se déclencher et que toutes sauf une sont mises en attente, soit parce que une seule règle demande en partie action l'exécution de plusieurs tâches, les tâches sont généralement empilées dans un agenda ; un agenda est une structure de données dont les éléments sont affectés de priorités ou plus généralement de prédictats qui rythment leur sortie.

La représentation fondamentale qui sert à DENDRAL tant à l'intérieur des règles que pour communiquer avec l'expert est celle du graphe de la formule développée des structures chimiques.

DENDRAL est utilisé quotidiennement par les chimistes de Stanford. Il a donné lieu à 25 articles dans des journaux spécialisés de chimie.

7.3.4.2 META-DENDRAL

Sujet : inférer automatiquement et de façon efficace des règles de fragmentation de molécules qui servent à DENDRAL. Les données sont des fragmentations types pour des échantillons de corps relativement simples.

Origine : le goulet d'étranglement dans DENDRAL était l'acquisition des règles, données de façon très morcelée par les chimistes (Buchanan 78). Il fallait donc obtenir plus vite un meilleur ensemble de règles.

Caractéristiques : Ici aussi la méthode combinatoire a été utilisée. Les graphes des corps donnés sont combinés de toutes les façons possibles, chaque configuration devenant le membre gauche d'une nouvelle règle. Le membre droit est constitué des pics probables dans le spectrogramme considérés comme intéressants d'après une statistique sur les échantillons. Toutes les règles engendrées sont alors testées et réarrangées selon quatre procédures :

- a) Oter les redondances.
- b) Agréger les règles qui ont les mêmes parties gauches.
- c) Si il existe une règle comportant une prémissse négative tenter de spécialiser la règle pour ôter cette prémissse.
- d) Essayer de généraliser des règles en conservant les mêmes prémisses positives et les mêmes résultats.

META-DENDRAL a retrouvé un ensemble de règles qui rivalise avec celui obtenu plus laborieusement par les experts. A propos d'un sous-domaine mal connu, META-DENDRAL a fourni un ensemble nouveau de règles, jugé excellent par les chimistes et qui a fait l'objet d'une communication dans une revue internationale de chimie (1976). Etendu aux spectres de résonance magnétique nucléaire, le système a rapidement formé, à partir de données en vrac très grossières, les règles plus complexes que l'on cherchait précisément à obtenir des experts (Buchanan 78).

7.3.4.3 Le programme de Waterman (1970)

Sujet : Apprentissage d'heuristiques au poker.

Origine : Waterman désire écrire un système expérimental intelligent où la séparation entre programme et heuristiques soit claire. De plus, le programme, au cours du jeu, doit être capable d'augmenter et de corriger lui-même son ensemble d'heuristiques.

Caractéristiques : Les heuristiques sont mises sous forme de règles de production qui sont petit à petit corrigées. Le programme est en outre capable d'en construire de nouvelles.

Une situation particulière dans le jeu est repérée par un vecteur de sept composantes : (valeur de la main, hauteur du pot, montant de la relance, bluff de l'adversaire (oui ou non), quotient pot/relance, échange de cartes (oui ou non), style du jeu de l'adversaire). Les heuristiques sont exprimées en fonction de ce vecteur :

Prise de décision et apprentissage dans le programme de Waterman

La structure de contrôle est ici particulière : les règles ne sont pas rangées en vrac mais dans un ordre imposé : la première unification réussie, en suivant cet ordre, remporte

la décision. La correction d'une erreur est de ce fait rendue assez complexe, tout ajout, toute modification entraîne des effets difficiles à prévoir.

La solution retenue par Waterman est la suivante : on forme explicitement la nouvelle règle qui décrit exactement la situation concernée, avec un nouveau membre droit qui corrige la décision fautive. Dès lors, de deux choses l'une : ou bien il existe déjà une règle qui peut être modifiée pour avoir le même effet que la nouvelle règle, ou bien il n'en existe pas. Dans le dernier cas, l'apprentissage est très simple : la nouvelle règle est placée juste avant la règle fautive. Le premier cas se divise lui-même en deux selon que la règle modifiable est placée avant, ou bien au contraire après, la règle fautive. Si c'est avant, on étend les champs de définition des valeurs symboliques des paramètres du membre gauche, de façon à ce qu'ils prennent en compte les valeurs de la nouvelle règle : l'ancienne règle a été généralisée. Si c'est après, le processus est l'inverse du précédent. Le programme particularise les règles situées entre la règle fautive (inclus) et la règle modifiable : les champs des paramètres sont restreints pour faire échouer toutes les unifications trop précoces.

Les dangers d'un tel schéma de contrôle sont multiples : difficulté de vérifier à la main l'effet d'un jeu de règles, création de règles redondantes que le système est obligé périodiquement d'éliminer, instabilité dans le comportement du programme en phase d'apprentissage. L'auteur rapporte cependant plusieurs séries d'expériences, avec ou sans professeur, qui montrent l'efficacité de l'apprentissage et donnent des résultats de qualité avec une vingtaine de règles seulement en phase finale.

Le programme possède un jeu changeant : il est difficile à cerner par l'adversaire ; mais, si ce caractère est une qualité au poker, ce peut être aussi bien un grave défaut dans d'autres domaines.

7.3.4.4 La famille MYCIN-TEIRESIAS

Sujet : Aide au diagnostic et au traitement, aide à l'entrée de connaissances, pour les infections bactériennes du sang.

Origine : E. Shortliffe (1974), R. Davis (1976), Université de Stanford, Equipe Buchanan-Felgenbaum.

Caractéristiques : Programme utile et effectivement utilisé, ce système possède en outre une connaissance de ce qu'il sait et de ce qu'il fait, qui lui permet d'expliquer son raisonnement, de répondre à des questions, d'apprendre et enfin de conseiller l'expert, le tout dans le jargon médical (voir paragraphe 7.2).

GUIDON : Ce système enseigne ce sur quoi MYCIN raisonne. Il est destiné aux étudiants en médecine à Stanford. La connaissance exprimée dans les règles de MYCIN est filtrée par un jeu de règles pédagogiques (Clancey 79). Mais alors que la recherche de MYCIN est aveugle et exhaustive, il convient d'enseigner au débutant ces règles progressivement.

Une stratégie de diagnostic doit être choisie : un mal de dent n'a pas la gravité d'une méningite. L'équipe de médecins informaticiens de Stanford a ainsi été conduite à découper encore les règles afin d'isoler la hiérarchie des entités cliniques et d'exprimer par ailleurs leurs liens. Un nouveau système de connaissances a été créé, il a pour nom **NEOMYCIN** (Clancey 81).

ONCOCIN

Edward Shortliffe écrit actuellement un système d'aide au traitement du cancer. La conception d'ONCOCIN diffère de MYCIN sur plusieurs points :

- Les patients doivent être suivis : pour chacun d'eux un protocole clinique est établi et mis à jour. Le temps intervient ainsi comme un paramètre fondamental. Les attributs ne sont plus des constantes mais des variables (ceci modifie profondément le moteur d'inférence : l'évaluation des prémisses n'est plus immédiate, voir paragraphe 7.5).
- Le domaine est si vaste qu'il faut classer les règles par contexte d'application ;
- Pour la même raison, le système doit en permanence contrôler la cohérence et la complétude du corps des règles (cf. paragraphe 7.6) ;
- Dans le but d'éviter des examens longs et coûteux, des règles par défaut tentent de donner des valeurs aux paramètres inconnus ;
- La stratégie de recherche n'est plus exhaustive mais tient compte du but.

7.3.4.5 SU/X et quelques autres

SU/X : Sujet : Identifier et localiser des objets émetteurs de signaux dans l'espace (Nil 78).

Origine : Le système est un essai lancé par E. Felgenbaum et P. Nil (1976) pour suivre des situations et des connaissances qui évoluent. La mémoire instantanée des faits est du type « tableau noir » utilisé dans le système HEARSAY de Carnegie Mellon university (Erman 75). Notons que les systèmes de production ne sont en rien générés, contrairement aux systèmes procéduraux, par une telle évolution dynamique des connaissances.

Caractéristique : Les membres droits des règles spécifient des actions qui sont en fait ici des appels de procédures : réévaluer tel paramètre ou remplacer telle hypothèse par telle autre. Les règles servent ainsi principalement à contrôler l'explosion combinatoire.

VM : (Vontllation Monitor) (Peigenbaum 79) est un système de production d'aide aux soignants en réanimation. Il reçoit directement les signaux des différents capteurs cliniques et surveille en permanence leur évolution. Il déclenche l'alarme en cas d'anomalie grave.

CRYSALYS : est un dérivé de SU/X qui cherche à inférer la structure de protéines à partir de cartes de densité d'électrons en trois dimensions qui proviennent d'analyses cristallographiques aux rayons X. Les molécules des protéines sont assez complexes pour que la méthode exhaustive de DENDRAL ou MYCIN ne soit pas valable et il faut donc une structure de contrôle plus riche (Engelmore 79).

MOLGEN : C'est un autre système expert créé par la même équipe. Le sujet est ici la biologie moléculaire : à partir des différents « outils » connus pour couper, lier, insérer, tuer des molécules d'ADN, des effets connus de telles modifications et de l'état de la technologie chimique, il s'agit de savoir, pour un but génétique que l'on se propose de trouver, à partir de quoi et comment l'obtenir. La difficulté principale est ici de faire collaborer dans la base de connaissances des règles de production traduisant des informations aussi diverses que : la biologie, la topologie, la chimie, la technologie de manipulation, la génétique (Friedland 79).

RITA : Ecrit à la Rand par une équipe composée de R. Anderson, R. Glllogly et O. Waterman est un modèle d'aide à la conception, (Anderson 76).

PROSPECTOR : Est un système expert en géologie (SRI 1977) pour évaluer l'intérêt des prospections minières (Duda 80), (Konolidge 79). PROSPECTOR fait à la fois appel aux règles de production et aux réseaux sémantiques partitionnés d'Hendrix (Waterman 78). Son réseau de règles est compilé sous forme de graphe (Cf. paragraphe 7.5.4). Il a permis de découvrir en Colombie britannique une mine de molybdène d'un valeur estimée à un million de dollars (1983).

DIPMETER : A été écrit dans le département de la recherche de Schlumberger Doll (Davis 81) ; il a pour objet l'analyse des signaux de relevés physiques du sous-sol (« logs »). Outre les règles de type MYCiN, la base de connaissances comprend des procédures d'extraction des formes significatives du signal. Le problème principal est de corrélérer symboliquement les résultats des mesures pour reconstituer la nature réelle et la position des couches géologiques traversées.

LITHO : Analyse de Logs chez Schlumberger à Clamart (A. Bonnet 82, J.G. Ganas-cia 83). **DRILLING ADVISOR (1982)** : Réalisé à partir du moteur KS300 Teknowledge pour la société ELF Aquitaine. Il est destiné à l'analyse des accidents de forages pétroliers.

CASNET : Possède un modèle de raisonnement un peu plus évolué que MYCIN pour non seulement diagnostiquer et soigner mais aussi suivre l'évolution des glaucomes et adapter en conséquence ses recommandations thérapeutiques (Kullkowsky 76). Dans la même lignée, nous trouvons encore les systèmes :

HEADMED : Psychopharmacologie (Felgenbaum 77).

INTERNALIST (Université de Pittsburgh) : Est un système expert pour la médecine interne. Un grand nombre de domaines et de maladies humaines sont déjà codés par plusieurs dizaines de spécialistes. Le système global comportera plusieurs milliers de règles regroupées en quelques 80 sujets particuliers.

SACON : « ingénieur-conseil » en mécanique (Michie 80).

SOPHIE : Aide à l'enseignement par ordinateur de la détection des pannes dans des circuits électriques (Anderson 79).

EL et NASL : (Sussman 75) et (McDermott 78) aide à la conception de circuits électriques.

Ces trois derniers systèmes utilisent des routines de simulation pour tester des hypothèses.

NUUDGE : Elaboration d'emplois de temps notamment quand le problème est mal formulé (Goldstein 77).

En outre, des systèmes utilisant des règles de production dans des programmes de jeux, comme dans le programme de Waterman au poker, sont apparus récemment : **PARADISE** pour le jeu d'échecs (Wilkins 79), ainsi que le système de Quinian pour le jeu de la carte au bridge ou Popescu (84), Faller (85).

A côté de tous ces systèmes conçus essentiellement pour effectuer un diagnostic à partir de faits initiaux, les règles de production servent par ailleurs d'autres objectifs :

- Compréhension de textes écrits (encyclopédie QUID) (Kayser 81), le raisonnement du moteur s'effectue avec un degré de finesse variable et contrôlable ;
- Compréhension de la parole continue, système **HEARSAY III** (Erman 81) ;
- RESEDA**, analyse de documents historiques du haut Moyen Age français (Zarri 81). Le système recherche les motifs et les intentions des personnages ;

– Conception de gammes d’usinage, système **GARI** (Descottes 81). Certaines règles expriment de simples conseils donnés par les spécialistes. Ces conseils peuvent entrer en conflit. Le système est à même de lever de telles contradictions par un schéma rigoureux et original ;

SAM (Gascuel 81) traite les hypertension artérielle, les accidents vasculaires cérébraux, les cancers du larynx.

TOUBIB (IBM France, Fargues 83) : aide au diagnostic en médecine d’urgence.

SPHINX (M. Fieschi 82, 83, 84) : Traitait initialement les douleurs épigastriques, orienté ensuite sur la diabétologie et le traitement des icteres. Il communique en français avec l’utilisateur (MEDIUM Joubert 81). Il est capable d’enseignement assisté intelligent (D. Fieschi 84).

PROTIS (Soula 83) : traite du diabète et des icteres en tenant un raisonnement incertain particulièrement efficace.

TOM(Cognitech84) : dérivé de EMYCIN pour l’étude des maladies de la tomate.

CESSOL (Ayel 84) : choisit les essais géotechniques pour étudier les caractéristiques d’un terrain.

CRIQUET (INRIA Sophia) : utilisé pour la classification des galaxies.

R1 ou XCON : Etude de configurations d’ordinateurs. C’est un système expert conçu chez Digital Equipment Corp, pour proposer des systèmes VAX 780 complets satisfaisant un ensemble de contraintes imposées (espace total, distances et places de contrôleurs, spécifications du client, etc). Il a été écrit dans le langage OPS (cf. Infra) par une équipe de 15 personnes. Il comprend actuellement 2 000 règles (McDermott 83). Il est totalement opérationnel et utilisé quotidiennement pour les multiples commandes de DEC.

Enfin quelques systèmes se distinguent par leur généralité ou leur capacité d’apprentissage.

7.3.4.6 Systèmes Généraux

EMYCIN : Conçu par Van Melle (Van Meile 1979), c’est le programme MYCIN « essentiel » ou « vide ». Il sert à l’acquisition des règles pour un programme expert de type consultant dans un domaine quelconque, il a par exemple permis l’écriture de SACON et de DART :

DART : Est un produit IBM (Bennet 81) qui est un « consultant » pour les pannes de matériel ou de système (initialisation, connexion d’un utilisateur en temps partagé). Il a été écrit en 8 mois par 5 spécialistes en maintenance et comporte 200 règles, les règles sont essentiellement les traductions des protocoles de communication et de leurs anomalies possibles ;

OPS : Issu de l’Université de Carnegie-Mellon : il diffère notablement d’EMYCIN par la présence de variables (OPS travaille non plus en logique des propositions, mais en logique du premier ordre) ainsi que par la stratégie de sélection de règles et les critères d’évaluation (Forgy 77) ;

TANGO : Ecrit à l’Université d’Orsay par M.O. Cordier et M.C. Rousset, est également un moteur en logique du premier ordre qui présente la particularité de travailler

sans base de faits : de nouvelles règles sont créées à chaque déduction et en outre TANGO utilise le but qui lui est donné pour guider continuellement, par un chaînage arrière, sa recherche.

PECOS : (Barstow 79), Traduit en LISP des algorithmes abstraits de manipulation des symboles, de tri, de graphes et d'arithmétique. Les représentations des objets, les techniques de programmation sont choisies à l'aide de quelques 400 règles et peuvent aboutir à plusieurs versions différentes du programme, qui sont évaluées et utilisées par un système plus général de synthèse automatique de programmes : PSI.

APE : Est un système expert en programmation : il écrit automatiquement des programmes en LISP à partir des spécifications abstraites des types de données et des algorithmes. Les programmes engendrés (tris, recherche binaire, etc.) peuvent avoir quelques 60 lignes de code et utilisent la récursivité. Les types abstraits incluent les listes, files et piles, les tableaux, les arbres.

L'espoir des auteurs est d'arriver à une nouvelle théorie de la programmation fondée sur la connaissance modulaire de règle de base.

TROPIC : Conçu par J.C. Latombo Clatombs 77), c'est un programme général d'aide à la conception : architecture, transformateurs électriques : il possède un mécanisme efficace de gestion des conseils et de retour-arrière intelligent en cas d'échec.

AGE : « Attempt to Generalize », de Nil et Alelo (Nil 78), inclut un algorithme de généralisation systématique des règles entrées.

ARGOS-II : De M. Cayrol, B. Fade et H. Farreny (Cayrol 79a), (Cayrol 76b), (Farreny 80), est un système général simulant la prise de décisions d'un robot, résolvant des problèmes. L'aspect logique de contrôle a été particulièrement étudié : les règles sont inhibées ou réveillées à volonté avec une utilisation systématique d'objets formels et une liaison par attributs efficace et originale.

CAMELIA (Vivet 84) : Il s'agit d'un S. E d'aide à la construction de théorèmes en calcul intégral, développement limités, algèbre, arithmétique, etc... Il est programmé en LISP et utilise le programme REDUCE pour les calculs de base. Son originalité est d'échauffer un **plan de résolution** pour tout problème qui lui est soumis à partir de métarègles stratégiques. En outre, il sait évaluer les prémisses des règles par nécessité ; par exemple dans :

```

si.....
si positif ( $f$ ) = vrai
si....
alors

```

où f peut être une expression formelle quelconque contenant en particulier des paramètres inconnus, la prémissse positif (f) devient pour CAMELIA un nouveau sous-but qui sera évalué par récursion.

Le travail de M. Vivet s'oriente maintenant vers l'utilisation de la métaconnaissance pour modifier les règles brutes afin de les adapter à la situation en cours.

GOSSEYN (Fouet 84) : utilisé en mécanique pour la conception de pièces estampées et la fabrication de carters moteur (2000 règles).

MUSCADET (Pastre 84) : Il s'agit d'un système expert dédié à la démonstration de théorèmes mathématiques pures : Théorie des ensembles, espaces vectoriels topologiques : domaines dans lesquels MUSCADET a prouvé plusieurs théorèmes difficiles. La

base de connaissances contient aussi bien des règles spécifiques aux différents domaines que des stratégies générales de démonstration et des connaissances mathématiques universelles. Tous les types de savoir (règles, méta actions, méta règles) sont interprétés par le même moteur. D. Pastre a écrit sa base de connaissances avec l'aide de mathématiciens (docteurs d'Etat). La constatation principale qui s'est dégagée à cette occasion est que les experts sont incapables d'énoncer des règles à froid : ils raisonnent sur des exemples, des cas particuliers, des dessins. Ils peuvent mettre beaucoup de temps à démarrer ou être incapables d'expliquer le « pourquoi » de la méthode qu'ils ont choisie.

Un exemple de théorème démontré par MUSCADET en 14 étapes (quelques 200 règles appliquées) est le suivant : « Si U est un voisinage de O , alors il existe un voisinage V de O , inclus dans U , étoilé et symétrique par rapport à O »

7.3.4.7 Systèmes avec apprentissage

Mentionnons tout d'abord le système **PONTIUS-O** de I.Goldstein qui simule le comportement d'un homme apprenant à piloter un avion.

Le travail de Lenat (Lenat 75), (Lenat 77), **AM**, qui « découvre » des concepts en mathématiques, a eu un certain succès. A partir de quelques concepts élémentaires. AM en engendre de nouveaux et juge leur intérêt à l'aide de fonctions d'évaluation générées par des règles de production. Il redécouvre ainsi de quatre façons différentes la multiplication : par substitution itérée, par répétition de l'addition, par analogie avec la numérotation des éléments d'un produit cartésien, par l'étude du cardinal de l'ensemble des parties d'un ensemble. Il retrouve également le concept de nombre premier (fig 7.7) et le théorème fondamental de l'arithmétique.

Nom : nombres premiers
Définition :
Origine : nombre-de-diviseurs(x) = 2.
Prédicat : premier(x) ssi (pour-tout z) ($\frac{z}{x}$ implique $z = 1$ ou $z = x$).
Itération : (pour tout x , $x > 1$) : Pour i = 2 jusqu'à $x - 1$: NON ($\frac{i}{x}$).
Exemples : 2, 3, 5, 7, 11, 13, 17, 19...
Cas particuliers : 2, 3,
Echecs aux bornes : 0 et 1.
Nombre de contre-exemples : 12.
Généralisation :
nombre, nombre avec un nombre pair de diviseurs, nombre avec un nombre impair de diviseurs.
Spécialisation :
factorisation unique, paire de premiers, somme de premiers.
Conjectures : Qoldbach, diviseurs extrêmes.
Analogies :
dualité : nombres à nombre-de-diviseurs petit et diviseurs d'un nombre.
Intérêt :
liaison avec la multiplication, la division et toutes les opérations liées à celles-là.
Valeur : 800

FIG. 7.7 – Une représentation de haut niveau

Le « concept nombre premier » dans le système AM de Douglas Lenat au cours d'une exécution. Les facettes « slots » cf. paragraphe 7.5) de tels concepts sont complétées à mesure par le système. En outre, le système est capable de créer nouveaux concepts de lui-même par généralisation ou spécialisation d'anciens concepts.

Chaque règle comprend en partie droite une explication qui donne la raison pour laquelle la règle est éventuellement déclenchée. Le système, qui progresse essentiellement à partir d'exemples qu'il fabrique et généralise lui-même, met en oeuvre de nombreuses heuristiques calquées sur le comportement humain et a, par nature, un raisonnement inductif.

D. Lenat a récemment écrit un nouveau système, **EURISKO** qui généralise **AM** en travaillant dans un domaine quelconque et en construisant lui-même ses propres règles à partir de méta-connaissances (Lenat 1983).

Le système **BACON-3** de P. Langley (Langley 79), est assez similaire. Il concerne la physique et redécouvre les lois de Kepler, Coulomb ou Ohm. Les « Adaptive productions Systems » (**APS**) de D. Waterman (1975) constituent des modèles généraux pour effectuer des opérations d'arithmétique, apprendre des mots, compléter des séries. La liste des règles est ordonnée et l'apprentissage consiste à insérer de nouvelles règles à une place judicieusement choisie. Notons toutefois que cet ordre imposé va à la fois contre le caractère purement déclaratif des systèmes de production et contre nos connaissances en psychologie chez l'homme. Notre savoir, qui est peut-être localement ordonné dans certains cas précis, ne saurait l'être dans sa totalité.

Mitchell, Banerji et al (Mitchell 81) ont écrit un système en intégration formelle ca-

pable d'améliorer ses performances avec l'expérience : trouvant d'abord une primitive par recherche exhaustive, le programme extrait de son arbre de résolution le(s) chemin(s) utile(s). Il particularise alors les simples règles de réécriture mathématiques qu'il a utilisées en précisant leur contexte d'application et les transforme ainsi en règles de production au sens habituel, il généralise ensuite en étendant toute fonction spécifiée dans ces contextes à toute une classe : sinus devient n'importe quelle fonction trigonométrique, etc.

Réciproquement, les règles employées à tort sont complétées pour être écartées à l'avenir sur le même contexte. Les prémisses peuvent devenir trop générales ou trop particulières. Elles sont modifiées à l'aide des exercices suivants. Les auteurs rapportent une amélioration considérable des performances après cet apprentissage.

ANA : De J. McDermott (McDermott 79) est capable d'apprendre et de retenir la façon d'effectuer une tâche nouvelle en raisonnant par analogie à partir d'anciennes méthodes. ANA simule un robot qui gère le magasin d'un marchand de couleurs : il construit sans cesse de nouvelles règles de production en transposant d'anciennes situations aux cas nouveaux rencontrés et ce tant qu'il n'y a pas d'erreur de cohérence flagrante. L'intérêt de ce système est de montrer une nouvelle fois que des mécanismes très simples - ici des mécanismes d'apprentissage - peuvent se greffer très facilement et très naturellement sur les modèles par règles de production de façon efficace.

Le programme de simulation ACT : Dû à deux psychologues, J.Anderson et P. Kline (Anderson 79) de Carnegie-Mellon, mérite une mention spéciale, puisqu'il est à la fois général et capable d'apprendre. Ce système a été créé pour simuler le comportement humain à partir d'expériences classiques de « formation de concepts » en psychologie.

La connaissance d'ACT apparaît sous deux formes : - Un graphe de propositions, semblable aux réseaux sémantiques classiques en intelligence artificielle pour la partie déclarative ; - un système de productions pour la partie procédurale. Les additions et modifications à l'ensemble des règles de production sont obtenues par quatre moyens différents : désignation, renforcement, généralisation et discrimination (Anderson 79).

PS : Le système « production system » - PS - de M. Rychener (Rychener 76) a permis de réécrire sous forme de système de productions des programmes aussi divers que :

- **STUDENT** de Bobrow qui résout des exercices d'algèbre posés en anglais ;
- **EPAM** de Felgenbaum qui répond à des tests sur des suites de lettres ;
- **GPS** de Newell, Shaw et Simon, qui effectue des tâches dans des domaines variés :
- un programme de Perdue et Berliner, qui joue des finales d'échecs ;
- celui de Moran, qui s'exprime en anglais dans un univers de cubes ;
- **SHRDLU** de Winograd qui résout des problèmes et dialogue en anglais dans le même univers.

Enfin, le paragraphe suivant va présenter une autre famille de systèmes experts construits à partir de la logique des prédictats du premier ordre et d'une méthode d'inférence particulière connue sous le nom de principe de Résolution.

7.4 Les systèmes experts fondés sur la logique du premier ordre

Un langage classique construit précisément pour modéliser le raisonnement humain existe : c'est celui de la logique mathématique. Effectivement une règle de production « situation → conclusion possible » n'est rien d'autre, sous un formalisme un peu différent, qu'un théorème : c'est un énoncé logique, qui porte sur des assertions dont l'interprétation est vraie ou fausse. La seule différence est que les règles de l'expert sont a priori toutes admises (axiomes), même s'il est impossible de les démontrer. Les règles de production sont ainsi très proches des règles de réécriture connues en mathématiques : manipulations formelles, monoïdes, grammaires, algorithmes de Markov. Il est donc séduisant, pour concevoir un système expert, de se placer délibérément dans le cadre formel, bien connu de la logique.

La majorité des systèmes présentés jusqu'ici travaillait en logique des propositions (Cf 3 : toutes les entités sont des constantes). Nous présentons successivement deux systèmes « essentiels » : PROLOG et SNARK, basés sur le formalisme de la logique du premier ordre (présence de variables) et quelques-unes de leurs applications. En réalité PROLOG, qui a été développé par Alain Colmerauer (1971) dans le contexte de la compréhension des langues naturelles, a été créé il y a dix ans, avant même le concept de système expert.

7.4.1 Le Langage PROLOG

PROLOG est un langage de programmation où l'unité n'est pas l'instruction, mais un théorème de logique du premier ordre (Cf ??). Le vocable premier ordre signifie que les théorèmes contiennent des variables, paramètres quantifiés universellement, qui peuvent donc être remplacés par n'importe quelle expression bien formée du langage. Les variables sont commandées par des prédictats d'ordre n . L'écriture $P(x_1, x_2, \dots, x_n)$, où les x_i sont des variables et où P est un prédictat d'ordre n , a alors pour interprétation : « $P(x_1, x_2, \dots, x_n)$ » est vrai (Colmerauer 71 et Roussel 75). Les prédictats P_i ne peuvent pas, quant à eux, être quantifiés dans PROLOG (quand ils le sont, on parle de logiques d'ordre 2, 3, ... (Huet 78)).

PROLOG est un interpréteur particulier des formules de la logique du premier ordre. Il travaille sur des prédictats, mis sous la forme normale disjonctive habituelle ; toute clause (disjonction de prédictats) s'écrit :

$$(\neg h_1 \vee \neg h_2 \vee \dots \vee \neg h_i) \vee (C_1 \vee C_2 \vee \dots \vee C_j)$$

ceci s'interprète naturellement comme :

$$h_1 \wedge h_2 \wedge \dots \wedge h_i \supset C_1 \wedge C_2 \wedge \dots \wedge C_j$$

Une caractéristique de PROLOG est de ne manipuler que les clauses de Horn, ou clauses qui ont au plus un littéral positif (à droite), c'est-à-dire une seule conclusion c_1 . Ceci interdit donc les conclusions disjonctives du type $C_1 \vee C_2$. Un littéral est un terme quelconque h_i ou c_1 d'une clause sous forme normale.

Diverses implémentations de PROLOG existent actuellement depuis la version en Fortran (Roussel 75), suivie par un interpréteur portable (Battani 73). PROLOG a été exporté principalement au Portugal (Coelho 80) à Edinburgh (Warren 77), en Hongrie

(Szeedi 77) et au Japon, où le plan « de cinquième génération » en fait le langage microprogrammé de demain. La machine « EPILOG », de Sansonnet et Alli, interprète déjà avec une efficacité remarquable le langage PROLOG (Toulouse, Université P. Sabatier, 1982).

Tous les interpréteurs de PROLOG sont en fait des démonstrateurs de théorèmes basés sur le principe de Résolution. Ce principe, dû à Jacques Herbrand (Herbrand 31), repris et programmé par J.A. Robinson en 1965 (Robinson 65), systématisé le raisonnement par l'absurde : pour démontrer que : $p \supset q$, le principe de Résolution consiste à prouver que la proposition : $p \wedge \neg q$ est contradictoire. La méthode est a priori complète (elle permet de démontrer tous les théorèmes d'un système formel donné), mais elle est lourde (combinatoire) et peu naturelle (nous ne raisonnons que rarement par l'absurde). Néanmoins, les interpréteurs programmés sont plus efficaces que nous...

En fait, chaque clause PROLOG peut être interprétée à deux niveaux différents :

- 1) de façon procédurale via l'interpréteur comme précédemment,
- 2) de façon purement logique ou déclarative comme simple énoncé d'un théorème de l'univers de travail. (Cf chapitre 3).

Exemple :

Définition de la concaténation (Gallaire 81). La concaténation de toute chaîne (X) avec la chaîne vide (NIL) est (X) elle-même :

$$+ \text{CONCAT}(\text{nil}, X, X) \quad (\text{la partie hypothèse est vide}).$$

La concaténation de la chaîne $U = X.W$, où W , par définition du point, est le dernier caractère de U , avec la chaîne Y est la chaîne $X.Z$ si Z désigne la chaîne « W concaténée à Y » :

$$+ \text{CONCAT}(X.W, X, Z) - \text{CONCAT}(W, Y, Z)$$

L'intérêt de programmer en PROLOG est donc triple :

- 1) Les clauses traduisent directement les assertions utiles (théorèmes).
- 2) L'interpréteur fournit un algorithme d'unification tout programmé.
- 3) L'arborescence de recherche (essais combinatoires d'unification des clauses deux à deux) est également gérée automatiquement (backtrack LIFO).

En outre, le principe de Résolution peut être guidé par une stratégie qui fixe une priorité sur les clauses et les littéraux candidats à l'unification (exemple, la « Lush » Resolution (Kowalski 79) Chapitre 3.3.6). Les clauses sont balayées en

suivant l'ordre d'arrivée et, à l'intérieur de chaque clause, les littéraux sont balayés de droite à gauche. Cette stratégie est complète.

La version implémentée à Marseille comprend une recherche en profondeur d'abord avec possibilité de retour arrière programmable, qui rend la procédure incomplète, mais plus efficace. (Ceci est réalisé par intervention du caractère « / » qui joue le rôle d'une trappe interdisant tout retour sur un littéral situé à sa gauche).

PROLOG permet ainsi, dans un formalisme simple, à la fois de décrire et de résoudre des problèmes. Il ne reste à l'utilisateur qu'à trouver les prédictats convenables pour énoncer son problème.

7.4.2 PEACE : un exemple de programme expert en PROLOG

PEACE écrit à Toulouse par M. Dincbas (1979), est un exemple de programme expert, écrit en PROLOG, pour l'analyse et la synthèse de circuits électriques.

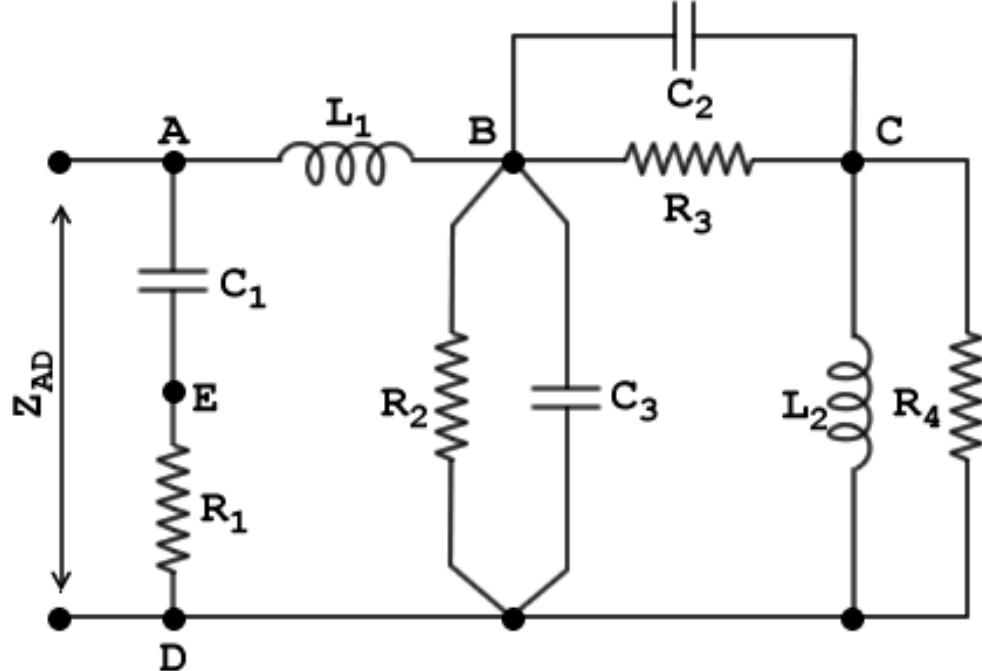


FIG. 7.8 – Un circuit électrique pour PEACE

Pour un circuit comme celui de la figure, avec des résistances, des inductances et des capacités données, ne comportant pas de source d'énergie, le problème de la recherche de l'impédance résultante entre A et D revient à démontrer le théorème :

$\neg \text{impédance } (A, D, * X).$

Le résultat, obtenu par l'absurde, sera donné par la valeur finale de la variable $* X$. Le circuit lui-même est représenté à l'aide de prédictats « égal », « branche » et « noeud-multiple » par des clauses de la forme :

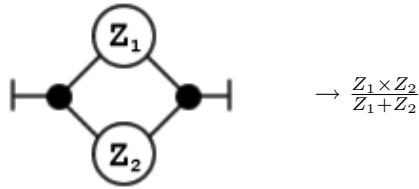
« branche (A, B, L'_1) » avec « égal $(L'_1, L_1\omega)$ »
et « égal $[C'_1, \frac{1}{C_1\omega}]$ »

ou encore :

« noeud-multiple $(B, 5)$ » .

Les connaissances fondamentales en électronique nécessaires à PEACE concernent les quatre transformations de circuits de la figure 7.9.

-
- Simplification parallèle :



- Simplification série :



- Transformation triangle-étoile :



- Transformation étoile -triangle : inverse de la précédente.

FIG. 7.9 – Transformations dans PEACE

PEACE a en outre besoin de savoir faire des calculs sur des nombres complexes. PROLOG ne connaît a priori que \mathbb{N} ; sa base de connaissances est étendue aux calculs sur \mathbb{Z} , puis sur \mathbb{R}^+ et \mathbb{R} (en précision de longueur paramétrée), enfin sur \mathbb{C} . La définition de l'addition dans \mathbb{Z} se fait par exemple à l'aide des clauses :

Plus $\mathbb{Z}(-u, v, -w) \Leftarrow$ inférieur-ou-égal $\mathbb{N}(v, u) \wedge$ moins $\mathbb{N}(u, v, w)$.

Soit :

$v \leq u$ et $u - v = \omega$ dans $\mathbb{N} \Rightarrow v - u = -\omega$ dans \mathbb{Z} .

Les règles de définitions des nombres complexes sont tout aussi simples.

Les transformations des circuits électriques sont décrites en PROLOG à l'aide de prédictats. La seconde par exemple devient :

```
« série ( $Z_1, Z_2$ )  $\Leftarrow$  branche ( $x, t, Z_1$ )  $\wedge$  branche ( $t, y, Z_1$ )
 $\wedge$  différent ( $t, y$ )  $\wedge$  noeud-simple ( $t$ )  $\wedge$  supprimer (branche ( $x, t, Z_1$ ))
 $\wedge$  supprimer (branche ( $t, y, Z_2$ )) plus ( $Z_1, Z_1, Z$ )
 $\wedge$  ajouter (branche ( $x, y, Z$ )) »
```

La stratégie interne à PEACE pour transformer le circuit est d'appliquer successivement les transformations dans l'ordre donné ci-dessus. Cet ordre correspond au nombre minimal de noeuds du réseau mis en cause. La détection des transformations licites à tout moment est faite par l'algorithme d'unification, et est fournie par PROLOG. Ainsi on peut, à chaque transformation, modifier de façon définitive la base de données.

On démontre que ce processus converge inéluctablement vers un circuit réduit à deux points. **La solution est obtenue de façon purement algorithmique.**

Dans un tel système, toute forme de connaissance - faits de base, procédures de Résolution, schémas de contrôle - est représentée de façon unique à l'aide d'une conjonction de clauses. Cette représentation unique est aussi une caractéristique fondamentale du système MECHO.

7.4.3 Second exemple :

MECHO et la résolution de problèmes en mécanique en PROLOG

MECHO résout, à l'aide de calcul des prédictats et de PROLOG, des problèmes de mécanique faisant intervenir des points matériels, des plans inclinés, des cordes, des poulies. Ces problèmes sont posés en anglais. MECHO est capable de contrôler ses règles d'inférence et sa recherche. Il utilise pour cela une métainformation qui concerne ces règles elles-mêmes. Le système a été développé à Edinburgh par A.Bundy et al. (1979).

La première tâche de ce programme est de parvenir à comprendre un énoncé en langue naturelle dans un cas où une approche par simples remplacements de mots (Bobrow(75), Charniak (78), Gelb(65)) ne peut convenir : le domaine est étendu, il n'existe pas de méthode générale, il faut une compréhension profonde de la situation physique pour simplement poser le problème.

La seconde tâche est d'apprendre ce qui n'est pas habituellement enseigné en mécanique : comment on résout un système d'équations, comment on construit un modèle, comment on contrôle la recherche. Il faut mettre au clair ce qui différencie l'expert du débutant : tous deux connaissent à peu près les mêmes faits et formules de base, mais le premier en fait meilleur usage : c'est le modèle général et la structure de contrôle qui diffèrent. L'équipe d'Edinburgh avait sous la main à ce sujet deux excellents points de départ. D'une part, un livre de D. Marples étudiait des protocoles d'ingénieurs de Cambridge sur la façon dont ils posaient des équations. Cette étude était suffisamment bien faite pour définir des clauses directement codables. D'autre part, A. Bundy avait lui-même analysé la résolution par l'homme des systèmes d'équations linéaires ou quadratiques. Il en avait déduit un ensemble de stratégies : « regrouper », « isoler », « attirer », exprimables en PROLOG. Il restait à énoncer et structurer tout le reste ; en particulier, à trouver les bonnes règles d'inférence, capables de reconstituer les longues chaînes de déduction depuis l'énoncé en anglais jusqu'à la solution.

7.4.3.1 Un exemple

« Deux particules de masse b et c sont reliées par une fine ficelle passant dans une poulie lisse. Trouver l'accélération de la particule de masse b . »

la procédure syntaxico-sémantique d'analyse, relativement simple, s'articule autour des verbes en s'aidant de « schémas » ou « scripts » de mécanique (Cf. paragraphe 7.6.2). Elle obtient ici un énoncé en 22 clauses unitaires :

est (particule, p_1)	masse (p_2, m_2 , temps – 1)
bout-droit (S_1, e_1)	mesure (m_1, b)
est (poulie, p_1)	donnée (m_1)
mifil(S_1, m_1)	est (fil, , S_1)
masse (p_1, m_1 , temps – 1)	contact-fixe (e_1, p_1 , temps – 1)
coeffrottement ($p_1, 0$)	contact-fixe (e_2, p_2 , temps – 1)
accélération	contact-fixe (p_1, m_1 , temps – 1)
(p_2 , accel, 90° , temps – 1)	masse ($S_1, 0$, temps – 1)
est (particule, p_2)	accélération
bout gauche (S_1, e_2)	(p_1 , accel, 270° , temps – 1)
est (période, temps – 1)	donnée (m_2)
mesure (m_2, c)	chercher (accel)

La méthode de Marbles cherche une équation qui exprime la quantité cherchée « accel » en fonction des données « m_1, m_2 » et « g », constante de gravité, ajoutée par MECHO au système. Au premier niveau, il retient les seules équations traduisant les forces qui s'exercent sur, par exemple, la particule p_1 et en déduit :

$$-m_1 \times g + \text{tension} - 1 = m_1 \times \text{accel}.$$

L'introduction de la nouvelle inconnue, tension – 1, a été forcée par une règle adaptée. Mais le programme sait que c'est effectivement une tension et sait par ailleurs qu'il lui manque une relation pour résoudre le problème. Il montre alors que la tension sur l'autre brin du fil est la même, écrit l'équilibre des forces sur p_2 et donne le résultat en éliminant tension – 1. D'autres exemples, notamment plus compliqués, ont été traités.

7.4.3.2 Description du système

Nous passerons ici sur plusieurs points pour ne retenir que ce qui concerne notre sujet :

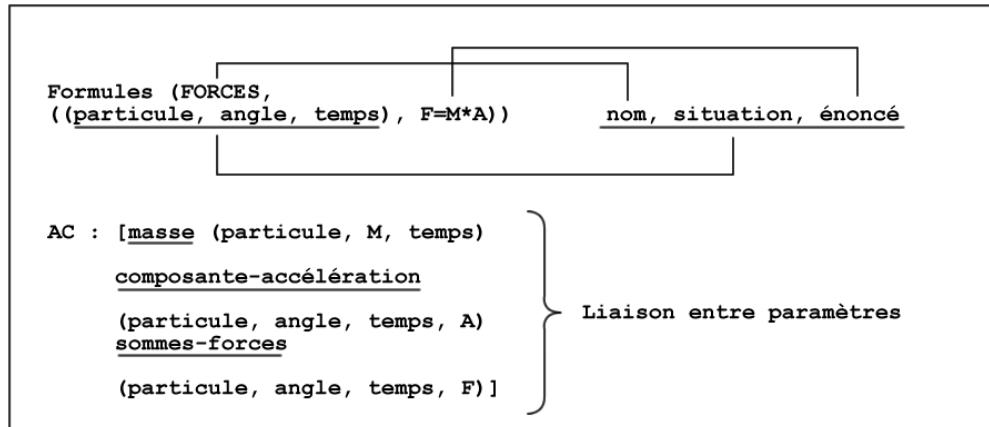
- A) La représentation des connaissances de l'univers physique.
- B) La logique de contrôle de la déduction et la résolution proprement dite.
- C) Exécution.

A) La base de connaissances physiques

Le système a une vue simplifiée du monde, qui correspond à la mécanique newtonienne. Il ne connaît que les objets à 0, 1 ou 2 dimensions. Chaque objet est repéré par son nom de façon biunivoque. Un point est un objet de dimension zéro qui peut être une particule avec une masse, ou un simple repère dans le plan. Une ligne est un objet de dimension un qui possède toujours deux extrémités distinctes. Beaucoup d'autres objets peuvent alors être décrits par des prédictats simples à partir de ces notions : une poulie, un homme, un poids sont des particules ; une corde ou un filin sont des lignes ; une barre, une planche sont également des lignes qui possèdent des propriétés physiques différentes.

Les relations entre objets sont traitées comme des prédictats dont le nom qualifie la relation et où les arguments précisent les objets en relation, dans un ordre et avec des conventions d'unité, de sens et de signes dont les circonstances de validité sont précisées en fonction de la valeur des paramètres d'appel.

Un exemple d'une telle formule est :



« AC » est un renvoi provoquant un calcul de type « à créer » (voir B), par le module de recherche des quantités nommées. On voit apparaître ici la structure fondamentalement récursive de la base de connaissances puisque « somme-forces » est, par exemple, donnée par une autre formule.

B) La logique de contrôle

MECHO possède une logique de contrôle écrite en PROLOG qui lui est propre et qui inhibe la stratégie de résolution normale de PROLOG : toute action de MECHO est progressivement décomposée en sous-but par l'une des 150 règles d'inférence de la base de connaissances. Chaque sous-but, quand il est créé, est marqué soit « D » pour « direct », soit « AC » pour « avec création », soit « SC » pour « sans création ». Ces marques provoquent des tentatives différentes, dans la stratégie de résolution, suivant le schéma de la figure 7.10.

Normalisation : Dans un premier temps, tout sous-but est, si possible, précisé à partir de suppositions licites en mécanique (si l'on cherche une accélération, c'est en général par rapport à la terre) ou bien encore traduit en prédictats « plus sympathiques » au système.

Inférence : MECHO cherche une règle dont le membre droit corresponde au but à atteindre. Les prémisses de cette règle définissent de nouveaux sous-but.

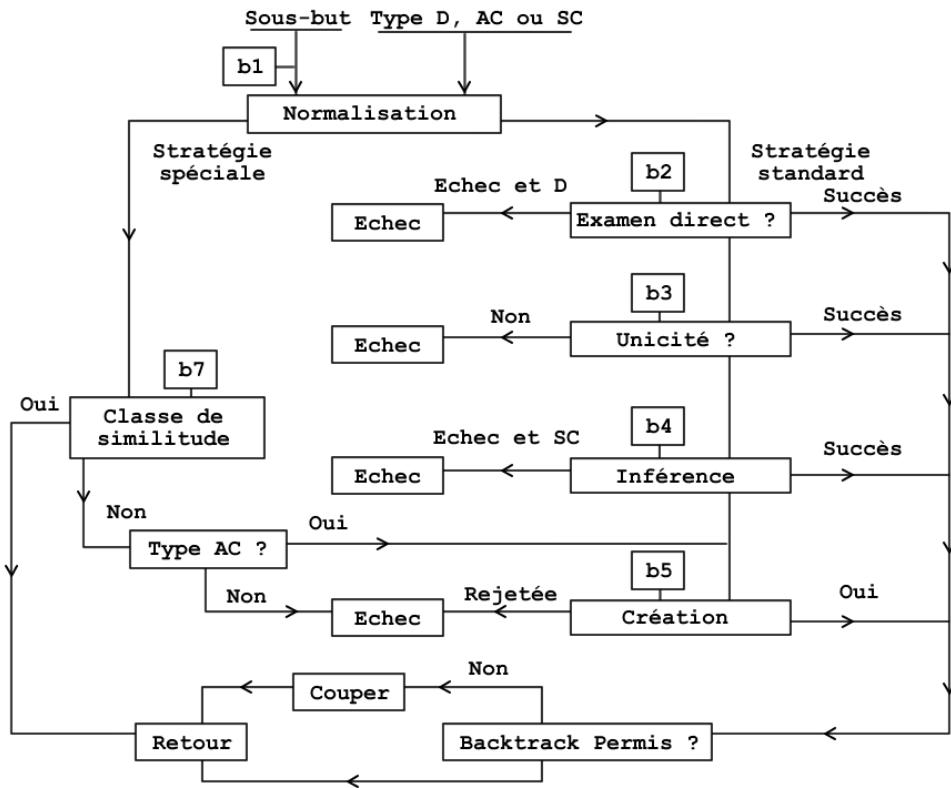


FIG. 7.10 – Logique des sous-buts dans MECHO

La recherche a lieu en profondeur d'abord avec un ordre figé des règles. Les clauses de MECHO contrôlent la création d'objets nouveaux, indispensables à la solution du problème, tout en évitant d'en introduire trop.

Classes de Similitude : MECHO connaît les propriétés des prédicats qu'il utilise. Ils peuvent être par exemple transitifs, symétriques, ou réflexifs. Ces prédicats qui sont alors leurs propres résolvants, sont regroupés en « classes de similitudes ». Ces classes sont isomorphes à des arbres représentés par des ensembles de clauses PROLOG.

La racine est le représentant unique de la classe et peut seule être unifiée. Pour l'égalité, on a ainsi l'axiomatisation :

$$\begin{aligned}
 &\text{« égal } (a, b) \leftarrow \text{représentant } (a, \text{racine}) \wedge \text{représentant } (b, \text{racine}) \\
 &\quad \text{représentant } (a, a) \leftarrow \text{est } (a, \text{racine}) \\
 &\quad \text{représentant } (a, \text{racine}) \leftarrow \text{arc}(a, b) \wedge \text{représentant } (b, \text{racine}) \text{ »}
 \end{aligned}$$

qui permet de ranger dans le même arbre tous les éléments égaux. Outre l'égalité, trois prédicats sont traités avec ce mécanisme : « même-position », « unités de mesure », « période-de-temps ». Ils concernent, respectivement les cas d'objets ponctuels occupant la même place à un moment donné, des quantités de même nature exprimées dans différentes unités, des périodes de temps ayant une partie commune.

C) Exécution

Première passe : dans un premier temps, MECHO ne considère aucune des équations ou relations qui introduiraient un objet nouveau (appels des seuls sous-buts « D » et « SC »). Cette contrainte est ultérieurement relâchée.

Prédiction : MECHO utilise l'information sémantique relative au monde physique pour décider si un but intermédiaire est acceptable ou pour prédire la nature de

la solution : une longueur doit être un nombre positif, un angle par rapport à une table doit être inférieur à 180° ; de même cette forme de connaissance intervient dans la manipulation d'égalités et dans la discussion des solutions.

La méthode de Marples : pour trouver une inconnue x , MECHO va successivement :

- a) décrire l'inconnue, i.e. , préciser sa nature physique (par exemple une masse) et donner les éléments qui lui sont liés (accélération, direction, temps) ;
- b) chercher les relations, i.e., faire la liste de toutes les formules possibles ;
- c) trouver une bonne équation, i.e., trier la liste précédente pour déterminer la formule qui s'adapte le mieux à la situation donnée, suivant la nature des paramètres connus ;
- d) tester si la formule retenue est effectivement indépendante des équations déjà fournies, afin de déterminer les autres inconnues.

Le calcul algébrique formel : outre les méthodes classiques basées sur les règles de réécriture, dont l'applicabilité est gérée par PROLOG lui-même, le module de calcul utilise l'information pragmatique en cours de résolution pour éliminer les situations physiquement impossibles ou d'équilibre instable (en particulier dans des problèmes de trajectoires de type montagnes russes), pour discuter les conditions d'inégalités suivant la valeur d'angles donnés ou encore pour prédire la forme d'une courbe (Borning 81).

L'intégration au système des règles d'inférence spécifiques à chaque situation pour résoudre tel ou tel cas a été effectuée pas à pas, essentiellement par examen des protocoles relevés par Marples, ou ceux des auteurs eux-mêmes, sur les problèmes soumis à MECHO.

7.4.3.3 Résultats

Le système tourne sur DEC10. utilise 80 K mots de 36 bits. La partie langage naturel est formée de 900 clauses, le module d'extraction d'équations de 600 clauses. le calcul algébrique de 200 ; 20 K sont utilisés pour l'interpréteur PROLOG.

Trois domaines ont été surtout testés : poulies, mouvements de particules sous accélération constante, mouvements de particules sur des trajectoires complexes. Seuls certains problèmes ont à ce jour été entrés en anglais, les autres l'ont été directement en termes de prédictats. Sont actuellement à l'étude (1985) : des problèmes de leviers, d'échafaudages, de projectiles, de moments d'inertie, de vitesses relatives, d'hydrostatique.

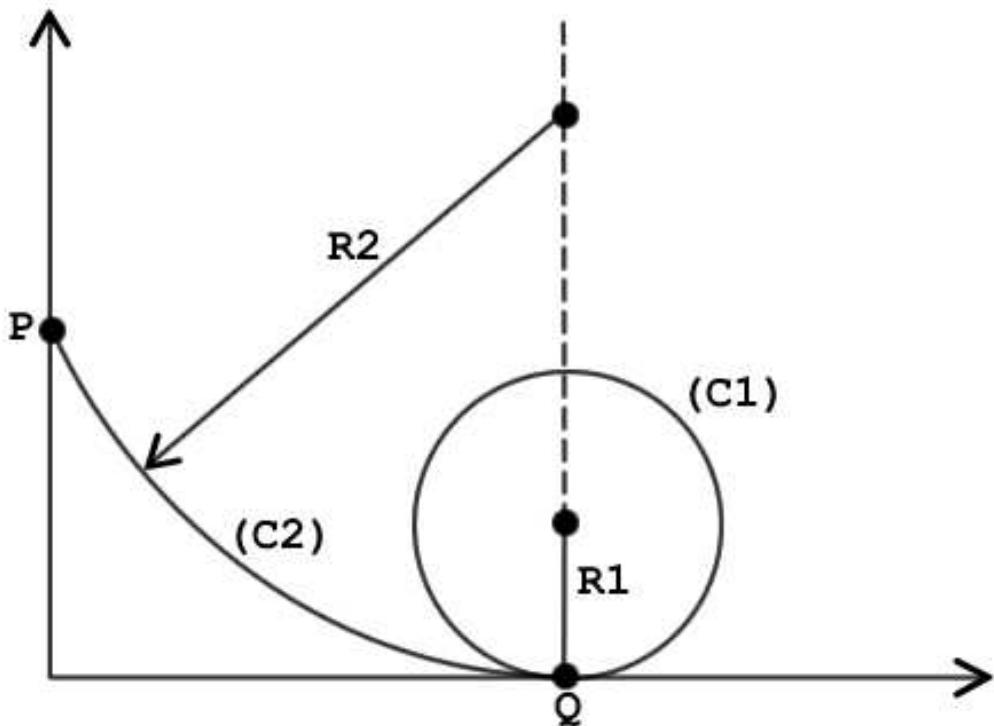


FIG. 7.11 – Exemple de la troisième famille

« Une particule part du repos en P , R_1 étant donné, quelle est la plus petite valeur de R_2 pour laquelle la particule fera un tour complet de (C_1) (C_1 et C_2 tangents en Q) ? »

Pour énoncer ce problème en logique du premier ordre, on découpe les circonférences en trois parties liées entre elles par leurs relations de position et c'est un ensemble d'une vingtaine de clauses unitaires qui joue ici le rôle de la figure.

Les auteurs relatent les résolutions complètes par MECHO de plusieurs problèmes complexes qui, résumées dans le langage habituel, peuvent aller jusqu'à prendre plus d'une page.

7.4.4 Intérêt de PROLOG et du Calcul des Prédicats

Le premier avantage des systèmes experts écrits dans le formalisme PROLOG est **l'uniformité de la représentation** : faits, règles d'inférences, procédures complexes, structures de contrôle, sont tous exprimés sous la forme unique de clauses conjonctives qui sont des disjonctions de littéraux. Le mécanisme de calcul, lié au principe de Résolution, est unique et entraîne la simplicité de programmation et la simplicité de la sémantique des programmes, facilement vérifiables. L'efficacité de PROLOG est, l'expérience le montre, favorablement comparable à celle de LISP. Le système expert MYCIN a ainsi complètement été reprogrammé en PROLOG avec une efficacité nettement accrue (Warren 77). En outre PROLOG offre des possibilités d'accès à l'arbre de la preuve, aux entrées-sorties, aux différentes parties des clauses et des littéraux, au mécanisme de retour en arrière dans l'arbre de preuve.

Mais l'intérêt primordial de PROLOG est, sans doute, d'autoriser l'écriture, en PRO-

LOG même, du contrôleur d'exécution, c'est-à-dire du processus de déduction lui-même qui travaille au niveau méta. Or c'est bien le problème fondamental dès que la base de connaissances devient importante. PROLOG permet en effet d'utiliser les clauses comme arguments pour des prédictats de niveau supérieur et offre ainsi des caractéristiques et des possibilités de calcul en logique du deuxième ordre. De nouvelles connaissances peuvent être créées ou supprimées grâce aux commandes AJOUT et SUPP.

7.4.4.1 Un langage de contrôle pour PROLOG

Hervé Gallaire et Claudine Lasserre (1979) proposent ainsi un méta-langage de contrôle de la déduction en introduisant dans PROLOG et à l'intérieur de la stratégie normale deux types de méta-règles. L'un, le type « Méta-L », servira à déterminer, dans la clause qui vient d'être obtenue, le littéral à unifier ; l'autre « Méta-C » sert à choisir la clause ancêtre parmi l'ensemble des clauses candidates. Les règles « Méta-L » se servent d'une évaluation locale, contextuelle, pour « geler » certains candidats ou introduire un retour-arrière obligatoire : l'idée est que tant qu'on n'a pas trouvé une installation indispensable. Il est inutile de continuer. L'objet des « Méta-C » règles est d'attribuer à chaque clause candidate une priorité qui est fonction de certaines conditions liées à l'unification.

Exemple :

$$\text{« Méta-C } (P, L, k, R) \Leftarrow C_1 \wedge C_2 \wedge \dots \wedge C_n \text{ »}$$

affirme que toute clause ancêtre, s'unifiant avec la clause $\neg P \vee L$ a la priorité k , dès que R s'unifie avec le littéral retenu à l'étape précédente et que C_1, C_2, \dots, C_n sont vérifiées. Ces méta-règles donnant à l'utilisateur toute latitude pour spécifier une stratégie adaptée à son problème particulier.

Le système METALOG écrit au CERT à Toulouse par Mehmet Dincbas procure à l'utilisateur un ensemble de commandes stratégiques, les méta-clauses. Ces prédictats de contrôle rendent le système PROLOG plus efficace et en font un véritable résolveur de problèmes.

7.4.4.2 Comparaison entre le calcul des prédictats en PROLOG et les systèmes de production

De toute évidence les clauses de Horn utilisées par PROLOG et les règles de production sont semblables. Plusieurs différences apparaissent toutefois à l'examen, qui tiennent toutes à ce que **dans PROLOG, la structure de contrôle est pré-programmée** (Cf. ??). Le principe de Résolution, avec une stratégie linéaire verrouillée, définit, seul et a priori, le type et l'ordre des essais à tenter. Cependant, il reste possible de modifier, sous PROLOG, cette stratégie de l'interpréteur. L'avantage essentiel de PROLOG est alors :

- 1) de bénéficier d'un algorithme général d'unification tout programmé.
- 2) de bénéficier également du mécanisme de gestion de l'arbre de recherche ET-OU, lui aussi fourni directement par l'interpréteur.

A la différence des systèmes de production, PROLOG augmente donc continuellement son nombre de règles (clauses), puisque chaque résolution en ajoute une nouvelle, jusqu'à l'obtention de la clause vide. Par ailleurs, les systèmes fondés sur le calcul des prédictats et le principe de Résolution confondent la base de règles et la base des faits. C'est une caractéristique de ces systèmes que de tout mettre au même niveau. En particulier, hypothèses et conclusions jouent le même rôle et la recherche se fait aussi bien depuis les unes, qu'à partir des autres, à rebours. Le sens de cette recherche est implicitement précisé à chaque résolution et peut varier à chaque fois. C'est un avantage sur les systèmes de production classiques qui n'interprètent, en général, les règles que dans un sens.

Si un schéma de raisonnement approché par coefficients de vraisemblance n'est pas au départ prévu dans PROLOG, on conçoit clairement qu'il est facile de doter l'interpréteur des capacités de calculs des C.V. du type MYCIN par exemple. Toutefois les clauses en PROLOG sont ordonnées : la modularité, la facilité de correction ou l'addition d'une unité de connaissance y sont moins grandes que dans les systèmes de production. **PROLOG reste avant tout un langage de programmation de haut niveau, mais de type procédural.** Il n'est pas lisible par le premier « expert » venu.

Il existe, bien sûr, une autre solution que PROLOG pour interpréter le langage de la logique du premier ordre : elle consiste à utiliser la **déduction naturelle** :

modus ponens : de p et de $(p \supset q)$, on déduit q ,
ou modus tollens : de $\neg q$ et de $(p \supset q)$ on déduit $\neg p$,

(à l'inverse de MYCIN qui travaillait en chaînage arrière et de PROLOG qui part à la fois dans les deux sens). En fait, le chaînage arrière partiel est indispensable pour guider la recherche et tendre vers le but, mais le raisonnement habituel, compréhensible par l'expert, se fait principalement par modus ponens.

Le système SNARK, présenté brièvement maintenant, est un effort dans ce sens. Nous décrivons SNARK (Symbolic Normalized Acquisition and Représentation of Knowledge) en donnant tout d'abord son système de représentation.

7.4.5 SNARK : un autre exemple de langage du premier ordre

Le langage SNARK est fondamentalement bâti sur celui de la logique du premier ordre et sur une expression des connaissances sous forme relationnelle. Contrairement à MYCIN, qui se limite au calcul des propositions, il fait donc un usage constant des variables (entités quantifiées universellement et partout substituables). Ainsi, la phrase : « $\text{nature}(x) = \text{homme}$ » signifie dans SNARK « pour tout x qui est un homme ». Le syllogisme bien connu associé, s'écrit exactement sous la forme :

« **si** $\text{nature}(x) = \text{homme}$ alors $\text{mortel}(x) = \text{vrai}$ »
et SNARK déduit du fait $\text{nature}(\text{Socrate}) = \text{homme}$, le fait :
 $\text{mortel}(\text{Socrate}) = \text{vrai}$

La même règle se déclenchera pour tous les x tels que $\text{nature}(x) = \text{homme}$.

Les relations utilisées sont par convention des relations binaires. Dans l'écriture des règles les variables figurent entre parenthèses.

Les propriétés peuvent être imbriquées, comme dans la phrase :

« âge (toit (x)) = min (150, âge (mur (y))) » .

Les objets ne sont généralement repérés que par l'ensemble des propriétés qui s'y rattachent, sans qu'il soit nécessaire de les caractériser par un nom. Ainsi, pour écrire une entité géologique G , placée sous une couche C , de nature « rift » et d'âge supérieur à 100 ans, on écrit :

$$\left| \begin{array}{l} \text{au-dessous } (C) = (G) \\ \text{nature } (G) = \text{rift} \\ \text{âge } (C) > 100 \end{array} \right.$$

Des théorèmes, de nature purement mathématique, peuvent être traduits dans SNARK. Ainsi la transitivité de « au-dessous » T :

$$\left| \begin{array}{l} \text{SI au-dessous } (x) = (y) \\ \text{SI au-dessous } (y) = (z) \\ \text{ALORS au-dessous } (x) = (z) \end{array} \right.$$

Ces caractéristiques font de SNARK un langage de la famille de PROLOG sans qu'il en ait certaines limites : outre le fait que SNARK n'utilise pas la résolution, mais la déduction naturelle, il n'est pas restreint aux clauses de Horn. L'ordre des prémisses, pas plus que celui des règles, n'a d'importance. Mais la souplesse de l'entrée des connaissances se paye par une menace d'explosion combinatoire. En effet, à un instant donné, dans une règle, la même variable est substituable par plusieurs objets de la base de faits. Un grand nombre de n-uplets sont ainsi candidats pour chaque évaluation. Cette menace d'explosion est maîtrisée par une stratégie d'évaluation dynamique des règles (l'idée est de partir de ce qui à un instant donné est connu et de le propager symboliquement (Cf. chapitre 8)).

SNARK possède ainsi les caractéristiques habituelles des langages déclaratifs aussi bien que celles des systèmes experts : modularité, connaissances énonciatives, et celles des résolveurs de problèmes : capacité de manipulation formelle, hypothèses et retours arrière, normalisation et unification d'expressions. Il est actuellement utilisé en géologie, en médecine, en archéologie, en mathématiques en enseignement assisté par ordinateur, en physique, en anthropologie, en contrôle de centrales nucléaires et en problèmes d'optimisation combinatoires pour le jeu de la carte au bridge. Décrivons plus finement une application particulière : le calcul formel de primitives.

Calcul formel de primitives en SNARK :

La base de connaissances est ici essentiellement constituée des règles heuristiques qui dictent la stratégie à adopter en face de chaque cas. Par exemple, si la quantité à intégrer est une somme (ou une somme élevée à une certaine puissance), alors il convient de décomposer l'intégrale. La règle correspondante s'écrit en SNARK :

Exemple 1

$$\left| \begin{array}{ll} \text{SOIT} & \text{expression-à-intégrer}(i) = (e) \\ \text{SI} & \text{somme}(e) = \text{vrai} \\ \text{ALORS} & i \leftarrow \text{intégrale (premier terme } (e)\text{)} + \\ & \quad \text{intégrale (deuxième terme } (e)\text{)} \end{array} \right.$$

Si e contient un arc trigonométrique, le changement de variable associé peut être envisagé :

Exemple 2

SOIT	expression-à-intégrer(i) = (e)
SOIT	terme(e) = (t)
SOIT	opérateur-tête(t) = arctrigo
ALORS	changement-variable (e) = (t)

Remarquons que :

- 1) L'appel de ces règles n'a lieu que par nécessité, c'est-à-dire lorsque l'une des prémisses au moins correspond à l'une des conclusions qui viennent d'être obtenues par le déclenchement de la règle précédente.
- 2) La variable (e) joue le rôle d'une variable globale. Tous les objets traités sont repérés et décrits par rapport à elle. Ceci correspond à un cas favorable qu'on ne retrouve pas dans toutes les applications.
- 3) Certaines prémisses jouent le rôle de définitions. Elles correspondent au SOIT. L'évaluation d'une règle commence donc par la partie SOIT, les SI ne sont testés que lorsque leurs arguments sont connus.
- 4) Il existe des variables (au sens mathématiques du terme) également dans la base de faits.
- 5) Certaines propriétés, en partie action, servent à passer des paramètres. Ainsi, l'expression « changement-variable (e) = (t) » est enregistrée telle quelle dans BF. Elle a ensuite éventuellement pour effet de déclencher la règle de changement de variable effectif qui s'énonce :
« si l'on veut faire le changement de variable $y = f(x)$, il faut pouvoir exprimer $(e).dx$ en fonction de y et de dy seulement »; soit en SNARK :

SI	changement-variable (e) = $f(x)$
ALORS	calculer dy = dérivée $f(x).dx$
	unifier (e, y, dy) (a)
	vérifier non-vu (e, x) (b)

où les deux actions correspondent respectivement à :

(a) : Appliquer systématiquement à l'expression les deux règles :

$$y \rightarrow f(x) \text{ et } dy \rightarrow f(x).dx$$

(b) : Vérifier qu'après cette opération la quantité à intégrer ne contient plus la variable x .

Par rapport à la programmation classique, on voit que :

- 1) Les règles n'ont nul besoin d'être ordonnées.
- 2) Les règles sont indépendantes : elles ne communiquent que par l'intermédiaire d'assertions sur la seule variable globale e - l'expression à intégrer - qui peut d'ailleurs changer en cours de résolution. Ceci est pris en charge par le démonstrateur :

$$\text{Cas } I = \int k \times e = k \times \int e \text{ ou encore } I = \int e1 + e2$$

- 3) Plusieurs règles peuvent correspondre à une même primitive. Par le jeu de priorités, liées à la complexité de l'expression sous le signe somme, c'est l'une ou l'autre qui sera d'abord choisie. Le démonstrateur revient sur les autres choix en cas d'échec.
- 4) Le jeu complet de règles pour le calcul de primitives comporte : 86 règles de stratégies, 368 règles de calcul en trigonométrie, exponentielles et logarithmes .
- 5) Un tel jeu de règles (stratégies d'intégration) est relativement long à rendre complet. L'expérience montre que des règles ont été oubliées, que d'autres ne sont pas suffisamment précises. Les modifications sont aisées, dans la mesure où le langage SNARK est lisible. Il est ici assez proche du langage du mathématicien. **On n'ose penser à quoi correspondrait le même travail en algorithmique classique.**
- 6) Le langage SNARK permet de décrire des règles très générales (comme les précédentes) aussi bien que des règles très précises ; ainsi par exemple :

$$\text{Cas } I = \int \left[\frac{1}{\sqrt{-3x^2+6x-2}} \right] dx$$

SI	<i>nature (e) = fraction</i>
SOIT	dénominateur (e) = (d)
SI	puissance fractionnaire (d) = <i>vrai</i>
SOIT	(t) = premier-terme(d)
SI	polynôme (t) = <i>vrai</i>
SI	degré (t) = 2
ALORS	identifier (t, ay ² + b)
SI	(a < 0) \wedge (b = 0)
ALORS	changement-variable (e) = (t) expression (t) = $\frac{b}{a} \times \cos\theta$

FIG. 7.12 – Règle d'interprétation dans SNARK

Rôle des variables dans SNARK :

Dans la représentation proposée ici, les variables instanciées renvoient aux enregistrements de la base de faits. Elles jouent, par là-même, quatre rôles différents qui correspondent à une standardisation, dans un même formalisme - la notation fonctionnelle - de quatre processus bien connus en programmation. Les variables servent en effet : 1) A indiquer tout renseignement (chaînage en liste) ; 2) A paramétriser les règles (passage d'arguments) ; 3) A évoquer un but (appel de sous-programme) ; 4) A gérer l'arbre de recherche (mise à jour d'une pile).

Détaillons ces points un à un :

- 1) Indexation : chaque variable de la base de fait est, par définition, instanciée. Elle correspond à une valeur qui repère un objet. Ce dernier n'est défini que par les relations actuellement connues, il se peut en particulier qu'il ne possède ni nom ni propriété caractéristique. Par ailleurs, tous les enregistrements portant sur la même propriété et réciproquement sur la même valeur, sont directement accessibles (on retrouve le schéma des bases de données relationnelles).
- 2) Paramétrage : Les variables des règles sont des variables formelles, substituables au sens mathématique. Les règles sont des théorèmes valides pour toute la famille des objets qui vérifient les hypothèses. Les appels et les passages de paramètres se font par valeurs puisque ce sont les valeurs des objets de la base qui décident du déclenchement.

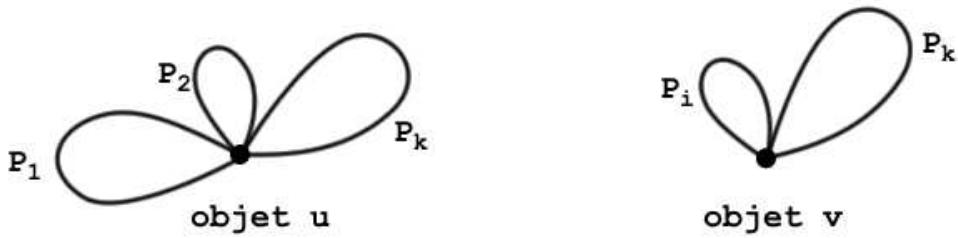


FIG. 7.13 – Un objet dans la base de faits de SNARK

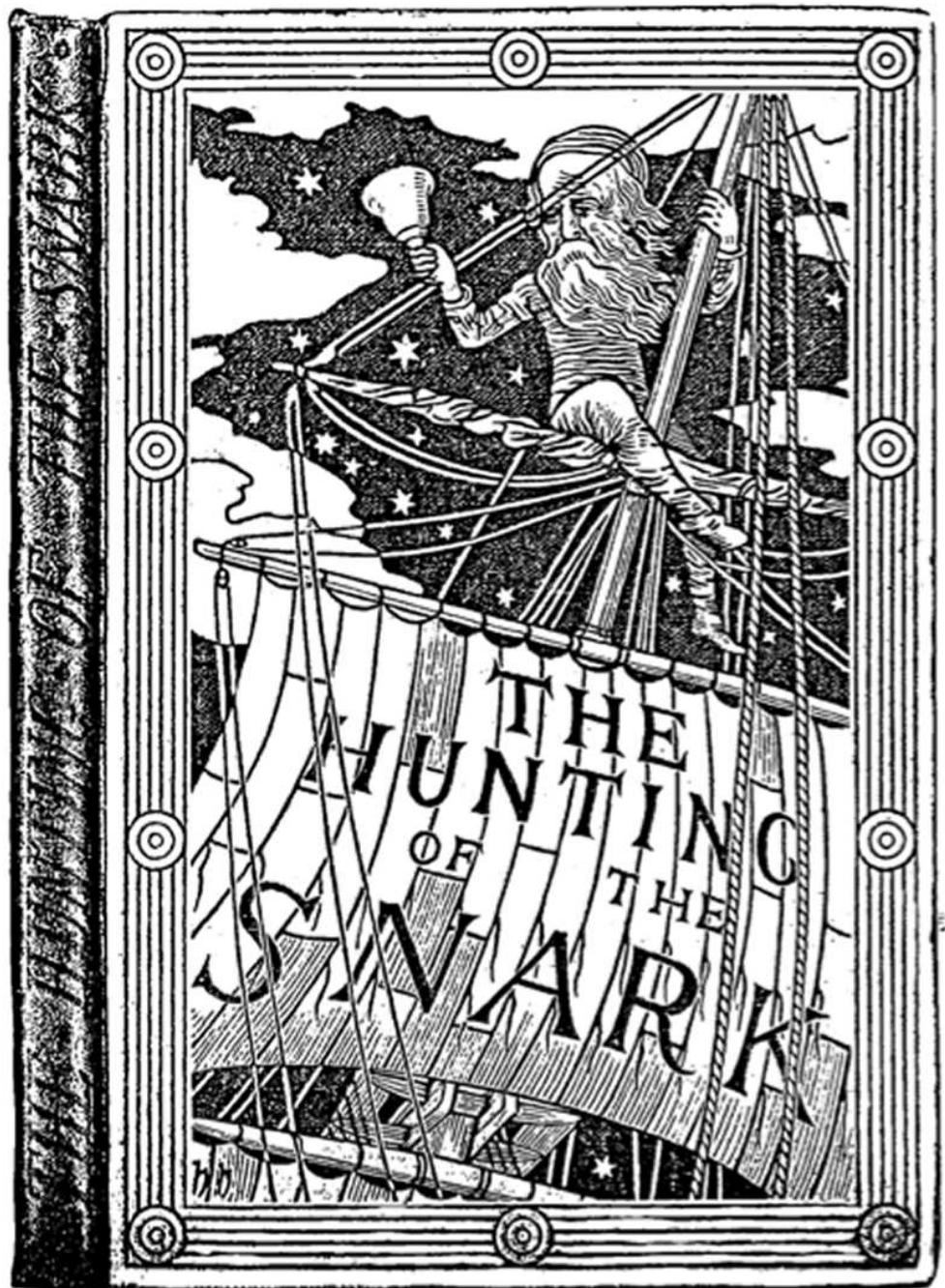
- 3) Appel de sous-programme : Une conclusion particulière d'une règle peut être associée, grâce aux variables, à l'évocation d'un nouveau but. Ainsi, l'enregistrement « changement variable (e) » permet la mémorisation, pour l'expression e , d'un but possible de changement de variable. Ce fait sera ensuite repris ou non par une autre règle, qui se chargera de le réaliser effectivement. On voit là un procédé original d'appel de procédure, puisqu'à tout instant, plusieurs buts-procédures peuvent être en compétition. **Le choix final est alors fait par des règles de plus haut niveau (métarègles)** qui évaluent les difficultés des différents buts.
- 4) Gestion de l'arbre de recherche : les problèmes traités obligent, par leur nature même, à d'éventuels retours en arrière lors de la résolution ; une règle s'est déclenchée trop tôt ou pour un mauvais jeu de variables ; le problème obtenu n'est pas soluble : il faut examiner les autres possibilités nous retrouvons une gestion d'arbre classique, traitée quelquefois par une programmation récursive. Ici la profondeur de la récursivité est complètement inconnue et peut être très grande. L'indexation par les variables permet d'effectuer simplement cette gestion : le choix d'un jeu particulier de déclenchement sert de paramètre d'indexation à tous les faits déduits à cet instant. Par exemple, l'expression e , après un changement de variable v , est marquée ev ; si, ultérieurement, la résolution de ev échoue, les informations sur ev sont oubliées et la recherche sur l'expression d'origine e reprend. Il est, là encore, possible d'exprimer, par métarègles, la stratégie de parcours de l'arbre de recherche et de donner par exemple la préférence à un instant donné, à l'expression de moindre complexité : **la gestion de l'arbre ainsi contrôlée de l'extérieur du programme**, « régime par tentatives » (Nilson 80) ou (Farennny 85).

A partir du démonstrateur particulier SNARK, qui travaille en logique du premier ordre, il a été ainsi possible, par exemple, d'écrire en quelques jours et de mettre au point rapidement un corpus de règles pour l'intégration formelle. Le temps de calcul d'une intégrale est en moyenne de l'ordre de la seconde sur IBM 370/168.

$$\text{Par exemple : } \int \frac{x^3 \times \text{Arcsin}x}{1 - x^2} \times dx.$$

Le gain en temps de mise au point par rapport à la programmation classique est évident. Un millier de faits peuvent être en moyenne inférés en une dizaine de secondes.

Une caractéristique importante du système est que dans chaque cas, le raisonnement, c'est-à-dire la séquence (éminemment variable) des règles utilisées, est facilement suivi. De plus, le système retrouve à la demande le sous-arbre de recherche qui a été utile pour prouver un fait quelconque à partir des connaissances initiales. Ceci fait de tels systèmes experts des **outils de choix en enseignement assisté par ordinateur**.



Après avoir exposé un certain nombre de réalisations participant toutes de l'approche déclarative des systèmes experts, c'est-à-dire utilisant toutes une base de connaissances indépendante du programme, nous comparons maintenant ces travaux à ceux effectués en informatique classique.

Les problèmes de la puissance et de l'efficacité de ces langages de représentation sont en particulier posés au paragraphe 7.5. Les voies actuelles de recherche y sont décrites.

Le paragraphe 7.6 tente de dégager les qualités et les défauts de l'approche déclarative des systèmes experts par opposition à la voie procédurale habituelle des informaticiens. L'expression du contrôle lui-même est l'objet du paragraphe 7.7 : **la métaconnaissance distingue fondamentalement les systèmes experts de systèmes de règles de réécriture**.

7.5 La controverse déclaratif/procédural

Cette controverse est vraiment née avec les premiers succès des systèmes experts, utilisant une base de connaissances mise sous forme de règles, de production du type :

MG Situation partiellement décrite	\longrightarrow	MD Conclusion envisageable
--	-------------------	----------------------------------

Par exemple, l'information :

« Tous les politiciens américains sont menteurs »

est entrée sous la forme :

$$\left| \begin{array}{lll} \text{SI} & \text{nature } (x) & = \text{politicien} \\ \text{SI} & \text{nationalité } (x) & = \text{amricain} \\ \text{ALORS} & \text{menteur } (x) & = \text{vrai} \end{array} \right.$$

Cette controverse a été également évoquée en programmation classique : c'est l'opposition entre applicatif et impératif, récursif et itératif, opérationnel et dénotationnel, statique et dynamique, fonctionnel et descriptif, opposition en somme entre procédures et spécifications. L'idéal est, bien sûr, de **spécifier** (un problème, un programme, un savoir, un savoir-faire) indépendamment des contraintes, des méthodes d'utilisation. Dans l'approche « déclarative » des règles de production et des systèmes experts, l'accès à l'information est alors obtenu par unification ou unification restreinte (dite « pattern matching » ou « filtrage »). (Waterman et Hayes-Roth 79, Nilsson 80, Farreny 80).

La réponse à « Y est-il menteur ? » s'obtient à l'aide de deux tests d'interrogation de la base des faits. De même « Y est politicien honnête » entraîne par unification : « Y n'est pas américain » .

Dans l'approche procédurale, au contraire, il faut prévoir les questions a priori et écrire une procédure pour chacune d'entre elles. Si quelqu'un est politicien, alors on teste sa nationalité pour savoir s'il est honnête, etc. Ainsi, une procédure exprime par essence un flot d'information ; elle traduit syntaxiquement comment transite la connaissance.

Les énoncés déclaratifs, quant à eux, expriment par définition une connaissance factuelle, de manière purement énonciative, indépendante de leur utilisation, donnée en vrac, de façon totalement modulaire. Ils traduisent une relation, une information sémantique. Ils répondent à une question de type quoi, insistons ici sur ce point qui est capital.

7.5.1 Critique de l'informatique « procédurale » ou « les langages de programmation ne sont pas des langages »

Le mot ordre a en français deux sens : le premier correspond à « l'acte par lequel le chef manifeste sa volonté (voir commander, imposer, ordonner) ». Le second désigne une « disposition régulière, une succession ou un enchaînement (voir suite, filiation) » (extraits du Robert). Il se trouve que ces deux acceptations du mot **ordre** correspondent aux deux caractéristiques fondamentales des langages de programmation actuels et, plus précisément, à la notion d'algorithme. Les langages de programmation permettent, en effet, l'écriture d'algorithmes qui sont :

- 1) exclusivement composés « d'instructions » (ordre 1) ;
- 2) définis par une séquence précise de celles-ci (ordre 2).

Il est ainsi impossible d'exprimer dans les langages de programmation disponibles sur le marché les phrases : « Le dollar est en hausse à la bourse de Paris » ou « Quand il fait beau le lac Léman est bien agréable », car elles ne correspondent à aucun ordre exécutable. En outre, elles sont - sans autre information - indépendantes l'une de l'autre.

Quelques informaticiens ont tenté de lever l'une de ces contraintes draconniennes, celle sur l'ordre des instructions, par l'introduction de nouvelles primitives (First i, Next i) ; ils ont montré que, dans certaines séquences types, les instructions pouvaient être permuteées sans changer l'algorithme. Les langages de programmation obtenus sont alors dits « sans affectation » (on ne réutilise pas deux fois la même variable). Les études d'actualité sur le parallélisme, les réseaux de Pétri, les schémas de programmes tournent autour de la même question.

Mais en réalité, le problème fondamental est ailleurs : **nous avons du mal à écrire des algorithmes, car cette notion même nous est étrangère**. Quand nous programmons, nous assemblons un véritable puzzle, dont nous ne connaissons que des morceaux épars : notre connaissance est effectivement modulaire et n'est disponible que par composants de quatre à six lignes.

Allons plus loin ; comment faire utiliser à un ordinateur une information comme : « Dans un circuit électrique, en courant continu, à la température ambiante, l'intensité I , la d.d.p. U et la résistance R sont liées par la loi d'Ohm : $U = RI$ » ?

Si une telle information peut être enregistrée telle quelle dans une base de données, toute la question est ensuite de pouvoir la récupérer et l'interpréter intelligemment, c'est-à-dire :

- 1) seulement à bon escient ;
- 2) dans tous les cas où elle est applicable. U en fonction de R et I , mais aussi R en fonction de U et I , I en fonction de U et R . En particulier, même si deux des valeurs U , R ou I sont inconnues, tout honnête homme écrira pour deux résistances en parallèle $\frac{R_1}{R_2} = \frac{I_2}{I_1}$.

On voit, en outre, que lorsque cette loi est utilisée, elle doit déclencher à son tour d'autres lois, qui sont en l'occurrence celles de l'algèbre et de l'analyse (simplification, normalisation). Ce point de vue met l'accent sur une ultime caractéristique des langages actuels de programmation : **ils supposent une information complète**. Toutes les données prévues dans un algorithme sont absolument requises pour le fonctionnement de celui-ci. Aucun paramétrage, aucun degré de liberté n'est a posteriori possible.

Or, un médecin qui effectue un diagnostic, un ingénieur qui conçoit un projet, un mathématicien qui cherche un théorème, travaillent avec une description seulement partielle des situations. Ils déduisent tout ce qui est possible, puis échafaudent, si besoin est, des hypothèses pour continuer. **Chaque situation est nouvelle**. Aucun schéma complet et préétabli ne dirige leurs réflexions. Leurs démarches sont complètement étrangères à l'algorithme habituelle.

Ces remarques expliquent qu'un champ immense du traitement de l'information échappe à la programmation classique : Il s'agit de tous les domaines, où l'homme manipule des **informations factuelles, des règles isolées, des stratégies et des savoir-faire partiels**. Toute cette connaissance est par **nature non algorithmique**. Certes, disposer d'un algorithme reste une excellente chose. Malheureusement, cette situation se présente rarement. Nés de besoins concrets (biologie et chimie) et des difficultés dans la mise au point de programmes, les systèmes dits « experts » tentent, nous l'avons vu, d'apporter une approche radicalement nouvelle à ces questions.

7.5.2 Les structures de contrôle dans ces systèmes experts

Les structures de contrôle des systèmes experts (SE) et des programmes procéduraux (PP) sont très différentes a priori. Dans un SE, le contrôle est « ouvert » : une règle peut être appelée à tout moment, sans autre influence avec les règles qui l'entourent que celle de la base des faits qui contient la description de la situation particulière traitée. Il y a alternance des cycles de sélection et d'exécution et après chaque cycle, tout est réévalué, l'ordre des règles recalculé dynamiquement.

Dans un PP, le contrôle est très localisé. Les outils de base du contrôle : ordre séquentiel de l'exécution, « SI-ALORS-SINON-, FAIRE-, REPETER TANT QUE », portent, à un instant donné, sur une toute petite partie des paramètres décrivant la situation. Le déroulement des instructions est bien moins lié aux données. Une petite variation en entrée change en général peu l'exécution d'une procédure. C'est l'opposé pour un SE. L'information contenue dans un SE est facilement accessible. La façon dont cette connaissance sera utilisée ne figure pas dans les règles, elle est implicite, c'est l'affaire des métarègles et de l'interpréteur. La situation est inversée dans un PP puisque le contrôle *y* est clair, tandis que la connaissance est dissoute dans le corps des procédures et des représentations utilisées. Les deux méthodologies, prises stricto sensu, s'opposent ainsi fortement.

Si l'utilisation de SE est nécessaire dans le cas de grandes bases de connaissances, il n'en reste pas moins que, pour des tâches spécifiques, bien localisées, le recours aux procédures s'impose. Or rien ne s'oppose à ce que les membres droits de certaines règles fassent **effectivement appel à des procédures**. Pour respecter l'esprit et pour conserver tout l'intérêt du SE, il est alors simplement nécessaire que de telles procédures ne s'appellent jamais mutuellement et soient ainsi toujours placées en fin de chaîne (c'est-à-dire en partie droite des règles).

	CONNAISSANCES	CONTROLE	RAISONNEMENT
Programmes procéduraux (PP)	codées (instructions)	codé (instructions)	boîtes noires
Systèmes experts (SE)	données règles et faits	donné sous forme de méta-règles	accessible, explicite, boîte en verre

FIG. 7.14 – Tableau récapitulatif : SE et PP

7.5.3 Règles de production et procédures : exemples

Les deux exemples qui suivent sont donnés dans l'intention d'illustrer le principe général de fonctionnement des systèmes de production, même dans des cas non favorables tels que l'expression de procédures essentiellement itératives. Ces exemples classiques montrent notamment qu'il est toujours possible, quitte à introduire des variables supplémentaires, de séparer concrètement les opérations en modules élémentaires exprimés sous formes de règles conditionnelles. A la lumière du second de ces exemples, où contrairement au premier cas, tout a été fait pour séparer complètement les règles et leur structure de contrôle, nous donnerons finalement la caractéristique fondamentale d'un système de production bien conçu.

Exemple 1 :

Inversion d'une chaîne de caractères par un algorithme de Markov.

(R1)	$\neq xy$	\mapsto	$y \neq x$
(R2)	$\neq\neq$	\mapsto	$\%$
(R3)	$\% \neq$	\mapsto	$\%$
(R4)	$\%x$	\mapsto	$x\%$
(R5)	$\%$	\mapsto	
(R6)	x	\mapsto	$\neq x$

FIG. 7.15 – Un système de règles de réécriture

Les symboles \neq et $\%$ sont des caractères de contrôle, qui ne peuvent être substitués : x et y , quant à eux peuvent s'unifier avec toute chaîne de caractères non vide. La liste des règles est ordonnée de haut en bas. La règle (R6), qui a une partie condition vide, s'applique à toute situation. La règle (R5), qui a une partie action vide, marque l'arrêt de la procédure.

MOT	\rightarrow	$\neq \text{MOT}$	(R6)
	\rightarrow	$\text{OT} \neq \text{M}$	(R1)
	\rightarrow	$\neq \text{OT} \neq \text{M}$	(R6)
	\rightarrow	$\text{T} \neq \text{O} \neq \text{M}$	(R1)
	\rightarrow	$\neq \text{T} \neq \text{O} \neq \text{M}$	(R6)
	\rightarrow	$\neq \neq \text{T} \neq \text{O} \neq \text{M}$	(R6)
	\rightarrow	$\% \text{T} \neq \text{O} \neq \text{M}$	(R2)
	\rightarrow	$\text{T}\% \neq \text{O} \neq \text{M}$	(R4)
	\rightarrow	$\text{T}\%\text{O} \neq \text{M}$	(R3)
	\rightarrow	$\text{TO}\% \neq \text{M}$	(R4)
	\rightarrow	$\text{TO}\%\text{M}$	(R3)
	\rightarrow	$\text{TOM}\%$	(R4)
	\rightarrow	TOM	(R5)

FIG. 7.16 – Inversion de la chaîne de caractère MOT par le système de Markov précédent

Cette procédure prend successivement chaque caractère de la chaîne et le place à gauche des caractères déjà échangés, puis ôte finalement les caractères de contrôle.

Exemple 2 :

On se donne les trois règles suivantes, où les symboles C1, C2, C3, désignent les chaînes quelconques de points, éventuellement vides [Post 36].

$$\begin{aligned} (\text{Règle f}) \quad & //C_1/C_2/ \rightarrow /C_2/ \\ (\text{Règle i}) \quad & /C_1/C_2/ \rightarrow /C_1/C_2// \\ (\text{Règle m}) \quad & /.C_1/C_2/C_3/ \rightarrow /C_1/C_2/C_2C_3/ \end{aligned}$$

Le symbole / est une constante non substituable. La chaîne d'entrée étant de la forme :

/ / /

p points *q* points

ces trois règles réalisent la multiplication des deux entiers *p* et *q* codés en systèmeinaire. Ici, les règles sont unifiées avec la chaîne d'entrée, dans un ordre quelconque. Elles sont, de plus, considérées comme des productions les unifications doivent se faire avec la totalité des membres gauches (Cf chapitre 3.4). Ainsi, avec pour entrée

la chaîne / / /, on aura successivement :

$$\begin{array}{ll} \text{par la règle i)} & : /.../....// \\ \text{par la règle m)} & : /.../..../..../(avec } C_3 \text{ vide), \\ \text{par la règle m)} & : /.../..../...../ \\ \text{par la règle m)} & : //..../...../ \\ \text{et par la règle f)} & : /...../ \end{array}$$

Note :

Depuis les débuts de l'informatique, nous avons été habitués à penser en termes de procédure et non en termes de réécriture. Il n'est donc pas étonnant que les systèmes de production nous soient a priori moins familiers. Mais l'exemple ci-dessus ne correspond-il pas à la manière de calculer des jeunes enfants ?

Dans les deux cas précédents, leur mécanisme paraît lourd : le formalisme des langages de programmation leur a été depuis préféré, avec raison, pour ces situations algorithmiques. En revanche, le formalisme des règles de production, unités granulaires de connaissance en vrac, prend tout son intérêt pour des énoncés de type énonciatif. Indépendants les uns des autres, comme c'est le cas des assertions, des heuristiques ou des règles de décision : théorèmes de mathématiques, règles de grammaire à la Chomsky, stratégies en diagnostic médical, savoir-faire en recherche minière, en génie chimique, en conception de fuselages, en électronique (cf. paragraphe 7.2). Et il n'est pas exclu que, comme chez l'homme sans doute, après plusieurs exécutions semblables, un paquet de règles soit retenu en bloc et figé comme une procédure.

7.5.4 Efficacité des systèmes experts

Le formalisme des règles de production n'est que la représentation externe de la connaissance. La plupart des systèmes « compilent » la base de règles pour obtenir une représentation interne efficace. La représentation interne a pour effet de structurer la connaissance en groupant en listes (au sens de LISP) règles et les prémisses. Deux réalisations en ce sens vont être décrites : POLITICS de J. Carbonell Jr. (1978), qui utilise une structure de règles imbriquées plus riche que le formalisme initial et le système PROSPECTOR (Duda 80) qui travaille en fait sur un multigraph. Les moteurs OPS et TANGO s'intéressent également par d'autres voies à optimiser la phase de déclenchement par des représentations internes des règles qui perdent en place mémoire pour gagner en temps calcul. L'interprétation, dans les deux cas, fait place à une compilation : la phase de déclenchement est facilitée. Il est important de noter que cette structuration de la connaissance est laissée à la charge du système puisqu'elle change profondément lorsque change la base de règles, ce qui est le cas commun.

Certains informaticiens ont par ailleurs pris récemment l'habitude de la modélisation par réseaux de Pétri (Pétri 62). L'analogie entre ceux-ci et les représentations internes utilisées dans les systèmes experts est profonde ; en particulier les réseaux de Pétri « à prédicats » ne sont qu'une redécouverte des systèmes de règles de production basés sur la logique du premier ordre (Cf. paragraphe 7.4). Soulignons enfin que la modularité essentielle des systèmes de règles est en elle-même un facteur d'efficacité. En effet, l'activation d'une règle se fait de façon associative et non dans un ordre imposé par une procédure : toute nouvelle conclusion n'évoque que les règles qui parlent d'elle. Les règles sont activées et éventuellement déclenchées en activant alors d'autres règles et ainsi de suite.

D'une manière générale, et, soit parce que plusieurs règles peuvent se déclencher et que toutes sauf une sont mises en attente, soit parce que une règle demande en partie action l'exécution de plusieurs tâches, les tâches sont généralement mises en attente dans un *agenda*. Un agenda est une structure de données dont les éléments sont affectés de priorités ou plus généralement de prédicats qui rythment leur sortie. Point n'est besoin de balayer les cas qui ne correspondent pas à la situation actuelle (instructions et tests de la programmation classique). Seuls ceux qui servent sont directement appelés par la base des faits, par les données. Les inférences, la résolution sont ainsi très différentes pour des données différentes. Ceci semble une condition indispensable à l'efficacité. **Ce n'est pas un hasard, si l'homme lui-même n'agit pas en déroulant des algorithmes** : même pour un cas trivial, comme la multiplication de deux nombres, nous utilisons chaque fois une méthode adaptée aux données : nombre le plus long « en

haut », sauf s'il comporte des 0, des 1 ou des chiffres répétés, prise en compte de la valeur de ceux-ci : petits chiffres en bas si -possible...

L'expérience montre par ailleurs concrètement, que la mise au point d'un corps de règles est agréable et rapide : il s'écoule quelques minutes entre la détection d'une erreur à l'exécution et la correction de la règle fautive et/ou l'adjonction d'une nouvelle règle. En revanche, dans un programme, une erreur détectée n'est pas toujours facilement corrigée à cause des « effets de bord » dus à la séquentialité : la correction effective peut demander, disons, des heures..., et ne peut être le seul fait de l'expert : **elle passe nécessairement par un informaticien, intermédiaire gênant et coûteux.**

Les règles structurées dans POLITICS

La structure rigide « SI-ALORS- », adoptée dès le début de cet article, peut être facilement complétée en « SI-ALORS-SINON- » et s'accommorder également de structures de SI imbriquées. C'est cette solution qui est envisagée dans ce paragraphe. L'avantage évident est, avant tout, de rapprocher les conditions du déclenchement, de l'action proprement dite et, ensuite, d'économiser l'écriture de prémisses communes à plusieurs situations. De tels systèmes ont été proposés par I.Golstein (1977) et J. Carbonell Jr. (1978).

POLITICS, le programme de Carbonell par exemple, invoque les règles d'après leur contexte en faisant appel à la notion de *script* (Schank 77), issue des recherches en compréhension des langues naturelles. Dès lors, le déclenchement s'effectue en plusieurs temps. Le but de POLITICS est de simuler le comportement humain avec ses préjugés et ses croyances, en face d'événements politiques. Les règles internes prennent la forme de celle de la figure 7.15.

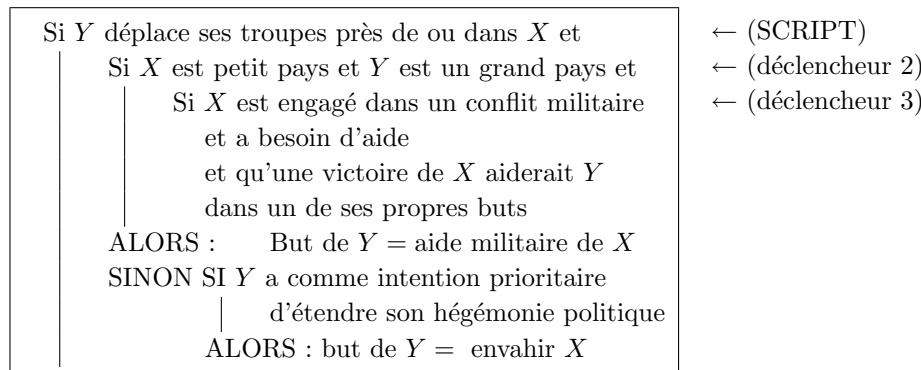


FIG. 7.17 – Une règle complexe dans POLITICS (Carbonnel 78)

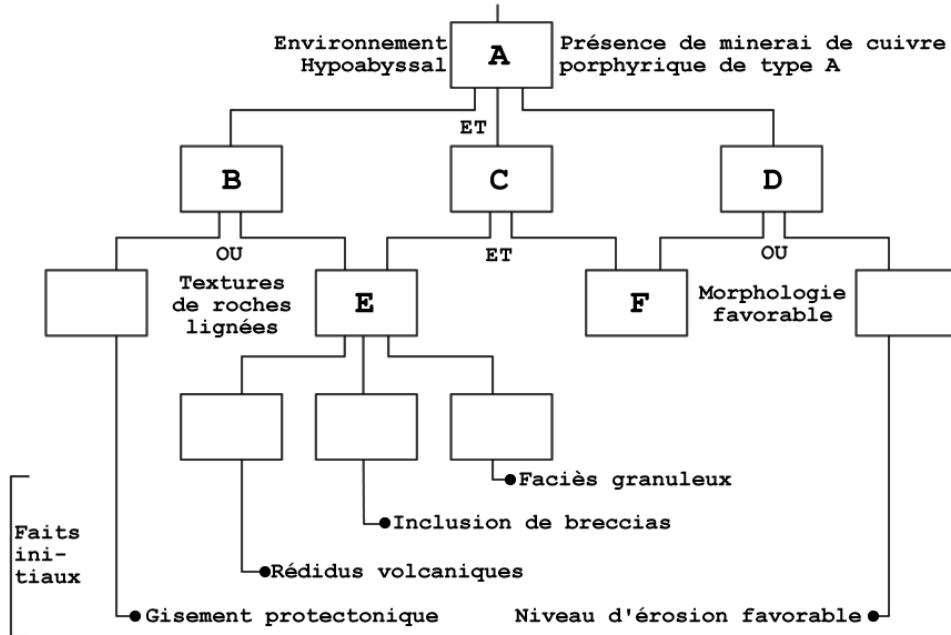


FIG. 7.18 – Extrait du réseau d'inférences formé par l'interpréteur de PROSPECTOR à partir d'un jeu de règles de production classique : les flèches regroupent les hypothèses qui permettent d'arriver à une conclusion.

Le réseau sémantique de PROSPECTOR

Une représentation interne quelque peu différente et, en fait, encore plus efficace, est celle utilisée dans le système d'aide à la recherche minière du SRI : PROSPECTOR. Le moteur d'inférences construit ici un graphe dont les sommets sont les propositions élémentaires. Une règle est alors un sous-ensemble d'arcs reliant des sommets : un sous-ensemble est marqué « ET » pour les prémisses d'une même règle ; il est marqué « OU » pour des conclusions identiques données par plusieurs règles (Cf. figure 7.16). Les déductions successives à partir des faits initiaux sont obtenues très simplement par propagation des pointeurs dans ce graphe. L'optimisation par rapport au but cherché (sommel sans successeur dans le graphe) est réalisée suivant les méthodes classiques en résolution de problèmes par intelligence artificielle : recherche dans les arbres ET-OU, algorithme de cheminement A^* (Nilsson 80 et chapitre 2).

Ces représentations internes sont alors très proches des *réseaux sémantiques* et des *trames* utilisés en compréhension du langage naturel (Minsky 75, Winograd 82, Pitrat 82) (Voir paragraphe 7.6.2).

Lien de ces systèmes avec les réseaux de Pétri

En 1962, C.A. Pétri proposa une représentation générale des processus sous forme de graphes, qui fut étendue par A. Holt en 1970. Ce dernier lui donna le nom de « réseaux de Pétri ». Ces réseaux sont des graphes orientés où les sommets sont soit des « places » (figurées par des cercles) soit des « transitions » (figurées par des barres verticales). Les places contiennent des « jetons ». Si toutes les places antécédentes d'une transition contiennent un jeton, alors la transition est dite active. Une telle transition est alors déclenchée : elle ôte un jeton à chaque place antécédente et dépose

un jeton dans chaque place successeur.

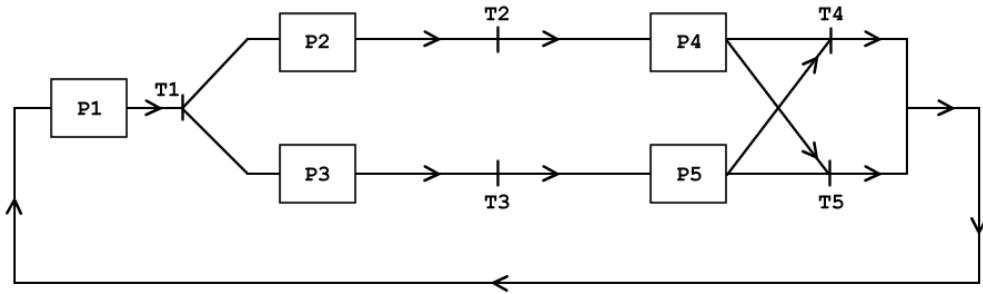


FIG. 7.19 – Exemple de réseau de Pétri : T1 peut se déclencher. P2 et P3 sont alors chargées. Ensuite le déclenchement de T2 et T3 charge P4 et P5 mais seule l'une d'entre elle pourra être active : puis le réseau se retrouvera dans l'état initial.

Il est clair qu'aussi bien les graphes de transition, que les automates d'états finis sont des cas particuliers de réseaux de Pétri. Ces derniers ont l'avantage de représenter, non seulement le processus lui-même, mais aussi son contrôle : jetons et règle de déclenchement, ils permettent ainsi de modéliser élégamment les problèmes de parallélisme et de synchronisation qui sont au coeur des processus de traitement de l'information et en particulier des systèmes de règles de production. Les réseaux de Pétri peuvent en effet avoir ici deux intérêts :

- 1) Mise en évidence des règles en conflit potentiel (possibilité de déclenchement simultané).
- 2) Représentation interne du jeu de règles pour l'interpréteur en vue d'optimiser le temps de résolution.

M. D. Zisman (1978) propose ainsi la modification suivante du jeu de règles donné en 7.5.3 pour retourner une chaîne de caractères : en partie droite de chaque règle est indiquée la règle à essayer après le déclenchement de la règle donnée.

$$\begin{array}{lll}
 (P1) & \neq \neq & \rightarrow \quad \wedge \quad (P4) \\
 (P2) & \neq xy & \rightarrow \quad \neq x \quad \wedge \quad (P2) \\
 (P3) & & \rightarrow \quad \neq \quad \wedge \quad (P1) \\
 (P4) & \neq & \rightarrow \quad \wedge \quad (P4)
 \end{array}$$

FIG. 7.20 – Jeu de règles (Fig. 7.15) modifié avec indications de transitions

On remarque qu'avec ce formalisme, un des caractères de marquage a pu être éliminé et que deux règles ont été gagnées. La représentation destinée à l'interpréteur sous forme de réseau est immédiate (Fig. 7.19). Il semble évident qu'une telle représentation, dans laquelle les flèches correspondent en fait au contrôle séquentiel des procédures habituelles et aux sauts conditionnels, doit être réservée au seul interpréteur : elle éloigne en effet de l'esprit même des systèmes de production, à savoir la modularité. Elle s'avère excellente en tant que représentation interne et a été utilisée avec quelques modifications sous le nom d'ATN (Augmented Transition Networks, Woods 72) pour l'analyse de textes écrits (Kayser 81, par exemple).

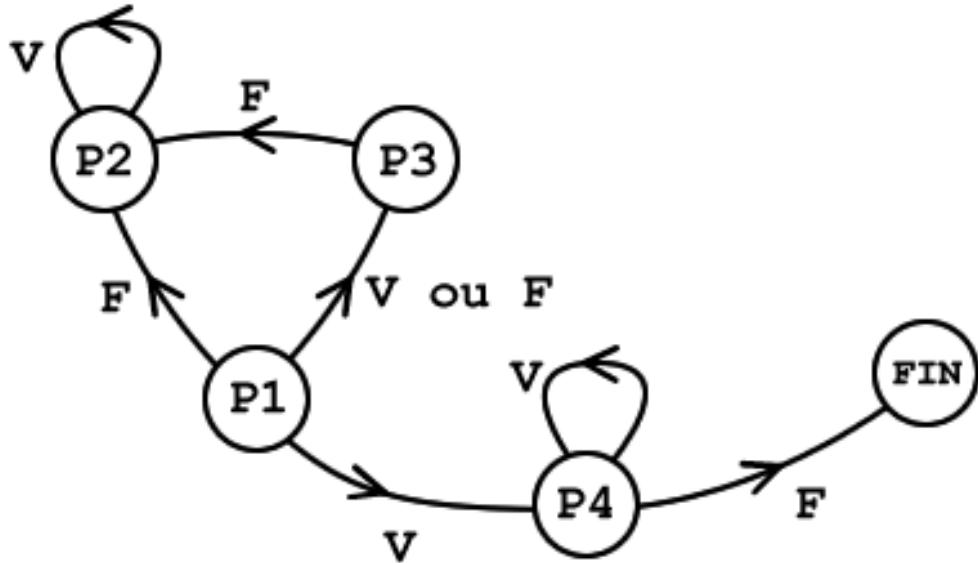


FIG. 7.21 – Représentation d'un système de règles de production sous forme de réseau de Pétri : les transitions ne sont pas indiquées par simplicité. Il y a deux sortes de jetons, vrais (V) ou faux (F), suivant la valeur de vérité de la partie gauche des règles.

7.5.5 Conclusion sur la controverse procédural/déclaratif

Il apparaît finalement que la différence fondamentale entre un programme usuel et un système de règles n'est pas le caractère plutôt procédural du premier et le caractère plus ou moins « déclaratif » du second, mais bien le fait que, dans le second, la structure de contrôle soit entièrement séparée des connaissances entrées sous forme de « règles » et donc que ces règles puissent être données en vrac. C'est bien en effet cette caractéristique, **le vrac**, qui est essentielle : elle définit les langages réellement déclaratifs, qu'ils soient fondés sur les règles de production ou d'autres logiques formelles. C'est la modularité absolue (unités de connaissance indépendantes, communications entre unités assurée par la seule base de faits, suppression de l'ordre, aux deux sens du mot, possibilité de remise en cause et de retour en arrière sur une règle) qui fait leur intérêt.

7.6 Les différents types de connaissances et leurs représentations

Les chercheurs en intelligence artificielle se fixent aujourd’hui pour but d’avoir des systèmes qui, d’une part, utilisent une grande quantité de connaissances transmises par les spécialistes et, d’autre part, sont capables de dialoguer et d’expliquer leur propre raisonnement. Ceci suppose une gestion efficace d’une base de connaissances importante et bien structurée, une distinction rigoureuse entre différents niveaux de savoir (Paragraphe 7.6.1), un ensemble de représentations convenables règles, prédictats schémas ou prototypes (Paragraphe 7.6.2) et un processus bien défini d’échange d’informations

entre les différentes sources.

Enfin, il faut que le **système sache ce qu'il sait**. Une telle méta-connaissance est, chez l'homme, d'un usage constant dans la vie de tous les jours : par exemple si, rencontrant un ami dans la rue, on ne sait plus son nom, on est malgré tout à même de dire qu'on sait qu'on a su son nom et qu'on l'a oublié. Ou bien au contraire, on sait que l'on ne l'a jamais su. Nous utilisons d'ailleurs continuellement des règles heuristiques importantes fondées sur cette méta-connaissance, telles que : « si c'était vrai, je le saurais ». Elles dépendent de deux paramètres fondamentaux : l'importance du fait concerné et notre propre degré de compétence dans le domaine. En fait, chaque connaissance chez l'homme implique immédiatement une méta-connaissance, liée à la façon dont nous rangeons cette connaissance avec les autres, comment nous la jugeons, pour quels buts elle peut être utile, à quelle famille elle appartient. Tous ces points sont développés dans les sections qui viennent à propos de systèmes informatiques réels.

7.6.1 Les différents types de connaissances

Huit grands types de connaissances nous semblent se dégager, qui concernent respectivement :

- a) *Les éléments de base, objets du monde réel.* Ils sont liés à la perception immédiate, que nous ne remettons pas en cause et qui s'ajoutent tels quels à notre base de faits.
- b) *Les assertions et définitions.* Elles portent sur ces objets de base. Elles sont a priori considérées comme sûres.
- c) *Les concepts.* Ce sont des regroupements ou généralisations des objets de base que chacun forme à sa façon. Comme dans les systèmes MECHO (Bundy 79) ou AM (Lenat 77). Ils sont liés à des exemples, à des contre-exemples, à des cas particuliers, à des concepts plus généraux ou analogues.
- d) *Les relations.* Elles traduisent aussi bien des propriétés élémentaires des éléments de base, que des relations de cause à effet entre concepts. De plus, ces relations sont marquées comme plus ou moins vraisemblables, plus ou moins corrélées avec une situation donnée. Soulignons à nouveau que la représentation des connaissances dans les systèmes experts est a priori proche des modèles utilisés dans les bases de données. Le modèle relationnel (généralisé) est ainsi à la base de OPS, PROLOG, SNARK ou TANGO.

Les doublets « attribut-valeur » (relationship-entity) bien connus dans les réseaux sémantiques, les frames et les scripts ne sont que l'expression la plus simple des relations binaires. Certains systèmes experts (INTERNIST, BAOBAB, SPHINX) utilisent déjà comme base de faits des bases de données existantes.

- e) *Les théorèmes et règles de réécriture.* Ce sont des cas particuliers de règles de production, qui ont la caractéristique d'être sûres. Les théorèmes ne sont d'aucune utilité sans les règles expertes pour les utiliser. L'existence explicite de ces théorèmes dans les systèmes experts fait toute la différence avec les systèmes de gestion de bases de données classiques (SGBD) où ceux-ci sont soit absents soit programmés. La modification ou l'adjonction de théorèmes nouveaux y est très difficile alors qu'elle est indispensable tant pour une bonne gestion structurée de la base de données que pour l'optimisation des réponses (Gajjaire 78).
- f) *Les algorithmes de résolution.* Ils sont indispensables pour accomplir certaines tâches : ils peuvent ou non avoir été appris. On passe en tout cas à une connaissance d'un type spécial puisqu'une suite d'actions se trouve être mémorisée en bloc

suivant un ordre dont l'importance est vitale, contrairement aux autres types, où les éléments d'informations peuvent être appris et emmagasinés individuellement.

Il ne semble pas que nous puissions retenir de longues procédures de résolution, faisant intervenir un grand nombre d'actions différentes. L'utilisation d'algorithmes purs est donc limitée à des cas très particuliers, numériques pour la plupart. Or l'homme doit traiter bien d'autres types d'informations. Il se trouve que l'ordinateur, bien plus qu'un simple calculateur, peut aider l'homme dans toutes les situations non algorithmiques.

- g) *Les stratégies et les heuristiques.* Ce sont des règles de comportement, innées ou acquises, qui permettent d'inférer des actions à envisager dans une situation précise. Elles utilisent les informations brutes en sens inverse du sens où elles ont été apprises : je sais que telle action produit en général tel résultat (information de type d) donc, si je veux tel résultat, je peux considérer telle action. L'homme en fait un usage constant que ce soit en perception, en formation de concepts, en résolution de problèmes ou en raisonnement formel.

C'est la nécessité de prendre en compte ce type fondamental de connaissance humaine qui est à la naissance des systèmes experts en intelligence artificielle.

- h) *La métacognition.* Elle intervient sans aucun doute à plusieurs niveaux, par paliers successifs. Il s'agit d'abord de savoir ce qui est su et quel coefficient de confiance lui accorder, quelle importance donner à une information élémentaire par rapport à l'ensemble des connaissances. En outre, il faut organiser les connaissances de chaque type et repérer quand et comment elles peuvent être utiles. Le paragraphe 7.7 sera consacré à cette question et montrera comment les systèmes actuels y répondent.

7.6.2 Les différentes représentations de la connaissance dans les systèmes actuels

Faisons brièvement la liste des procédés aujourd'hui les plus utilisés. Leurs avantages réciproques sont surtout techniques. La différence fondamentale concerne la facilité de modification de la connaissance. Plaçons-les (un peu brutalement, mais ceci n'est pas inutile pour fixer les idées) dans un tableau vertical en allant du plus procédural (le plus figé, le plus structuré) au plus déclaratif (le plus ouvert, le plus libre, le vrac) (Figure 7.22).

Automates finis, programmes, calcul des prédictats et systèmes de règles de production sont, rappelons-le (paragraphe 7.5.2), équivalents d'un point de vue théorique, puisqu'ils permettent tous de se ramener, en principe, à une machine de Turing universelle : la commodité d'emploi est cependant très différente de l'un à l'autre.

« Frames », « scripts », « réseaux sémantiques » sont des formalismes ad hoc pour les besoins présents en intelligence artificielle : ils sont très utilisés en compréhension du langage (Charniak 78, Cordier 79, Pitrat 82).

Les *frames* (ou prototypes) sont des structures de données complexes proposées par M. Minsky en 1975. Un frame ou prototype représente une situation typique comme l'attente dans un aéroport ou la participation à un dîner de famille. Un frame comporte des *places* (« slots ») pour les objets qui jouent normalement un rôle dans la situation. Une place peut elle-même renvoyer à un autre frame. De plus, il contient des informations sur la façon de l'utiliser (remplissage des places, valeurs par défauts,

activation), sur ce qu'il convient de faire dans les cas anormaux par rapport au cas type, ou par rapport aux frames « voisins » .

Un *script* ou schéma décrit un scénario stéréotypé et la suite d'actions des personnages mis en cause. Les scripts sont liés à la culture courante et sont indispensables pour comprendre des phrases comme : « J'entrai dans un restaurant, la serveuse m'apporta le menu » (Schank 75). Ils peuvent faire appel à d'autres scripts et s'intéressent, plus que les frames, à l'aspect dynamique de la connaissance.

Les *réseaux sémantiques* sont des graphes qui sont souvent des agrégations des deux entités précédentes et dont l'objet est de décrire les éléments de l'univers et leurs relations, ils ont bien des points communs avec les modèles des bases de données relationnelles.

Les dessins, les graphes, ont été effectivement utilisés en machine pour démontrer des théorèmes (Buthlon 75, Pastre 78 et chapitre 7) ; ils servent essentiellement d'aide à la preuve (cf chapitre 2) en court-circuitant des étapes, en mémorisant le problème de façon globale et efficace, en guidant l'introduction d'éléments nouveaux.

PROCEDURAL « FERME »		Texte	Référence
	1. Automate fini	5.1.	(Markov 54)
	2. Programme	5.1.	(Meyer 78)
	3. Scrip (schéma)	5.4., 6.2.	(Schank 77)
	4. Réseau sémantique	5.4.	(Mc Carthy 77)
	5. Frame (prototype)	5.4., 6.2.	(Minsky 75)
	6. Graphe, réseau	5.4.	(Pétri 62)
	7. Spécification formelle	6.1.	(Germain 81)
	8. Calcul des prédictats	4.1.	(Kowalski 79)
	9. Théorèmes, règles de réécriture	6.1.	(Huet 78)
DECLARATIF « OUVERT » « VRAC »	10. Règles de production	2.3.	(Shortliffe 76)
	11. Phrases en langage naturel naturel		(Pitrat 81)

FIG. 7.22 – Représentation de connaissances

7.6.3 Accès et modes d'utilisation des connaissances

Il y a quatre façons d'utiliser une information simple de la forme :

$$p \supset q$$

comme par exemple « tout homme est faillible » .

L'utilisation la plus immédiate consiste à se servir de l'implication dans le sens direct pour affirmer :

- 1) « Une entité est faillible dès que je peux prouver que c'est un être humain » . Une seconde utilisation qui passe par une métaconnaissance et inverse le sens de lecture est :
- 2) « Si Je veux prouver que quelque chose est faillible **un sous-but possible** est de montrer que c'est un être humain » .

Ces deux premières formes utilisent le modus ponens ; deux formes duales existent en prenant la négation de la proposition q et le modus tollens ;

- 3) « Si une entité est infaillible, ce ne peut être un homme » .
- 4) « Si Je désire prouver que quelque chose n'est pas un homme. Je peux tenter de montrer qu'elle est infaillible. »

Remarquons que le principe de Résolution, et donc en particulier le langage PROLOG qui l'utilise fondamentalement, fait, par construction, ces quatre types d'inférences sous un aspect formel unique.

Par ailleurs, d'un point de vue pratique, une information est utilisée en fonction de paramètres qui mesurent sa vraisemblance. Tout système de traitement de l'information doit pouvoir en effet répondre à des questions telles que :

- 1) « Quelle fraction de l'affirmation A est vraie ? »
- 2) « Quelle confiance un expert accorde-t-il à la conclusion ? »
- 3) « Quelle est, en moyenne, la probabilité d'arriver au but en utilisant A ? »
- 4) « Dans quelle mesure A doit-elle être préférée à autre chose ? »

Plusieurs systèmes, fondés sur les règles de production, utilisent déjà, de façon opérationnelle, des processus de raisonnement approché qui prennent en compte de tels paramètres.

Des systèmes utilisant les règles de production ont été décrits dans les paragraphes [7.2](#) et [7.3](#). Le paragraphe [7.4](#) a montré par ailleurs les différences possibles entre ces systèmes et l'approche des mêmes problèmes par la logique du premier ordre. Le prochain paragraphe fait le bilan des qualités et des défauts de ces modélisations de la connaissance.

7.6.4 Avantages et inconvénients des systèmes de production

Nous commencerons par les inconvénients. Il y en a trois principaux qui tiennent tous à une certaine rigidité qui est la base même des formalismes utilisés. Ils concernent respectivement la conception, l'énoncé et l'utilisation des règles (Cf. aussi la conclusion de ce chapitre au paragraphe [7.8](#)).

Inconvénients :

- 1) Difficulté de concevoir une règle de production correspondant à un élément de connaissance :

Il faut que le domaine concerné ait déjà été suffisamment étudié pour que les bonnes primitives aient été dégagées et que le niveau de détail ne soit pas trop fin : sans ce présupposé, il faut en effet une règle pour chaque situation et on voit mal comment l'homme pourrait aisément transmettre une telle masse de savoir informé.

- 2) Difficulté d'écrire une règle :

Le format unique : « SI-ALORS- » entraîne une certaine lourdeur dans les expressions des membres gauches et une certaine répétition des mêmes prémisses pour des situations semblables. Il est ainsi malaisé d'exprimer des règles complexes. Mais nous avons vu ([7.5.4](#)) que cette syntaxe rigide, sa lourdeur occasionnelle mise à part, présente bien des avantages. « Piéger les connaissances » reste en tous cas le problème majeur lors de l'écriture d'un système expert.

3) Difficulté d'utilisation :

Le dernier point n'est pas lié aux règles elles-mêmes, mais à leur système de communication unique et réduit à travers la base des faits. Cette base des faits joue le rôle d'une mémoire à court terme qu'elle simule d'ailleurs réellement dans le système PSG de Newell (Newell 75) en « psychologie cognitive ». Elle interdit que les règles s'appellent directement l'une l'autre et est malcommode dans l'exécution d'algorithme au sens habituel. En revanche, c'est par elle que le système sait, à chaque instant, précisément ce qu'il fait et ce qu'il sait.

D'une façon générale, l'examen des avantages intrinsèques de la programmation par règles vont faire apparaître ces inconvénients comme foncièrement nécessaires.

Avantages

1) Modularité :

Chaque unité d'information peut ici, par construction, être ôtée, changée, ajoutée sans qu'il soit besoin de modifier les autres. La connaissance est donnée en vrac comme dans un dictionnaire ou une encyclopédie. L'expérience montre que c'est une façon naturelle pour un expert de fournir sa connaissance.

Notons, que de façon étrange, de nombreux auteurs américains et particulièrement Davis, Newell, Quinlan, Stefik, Waterman dérogent à cette règle. Ils résolvent alors assez mal les problèmes de remaniement.

2) Modifiabilité :

C'est le caractère corollaire du précédent. Si l'on ajoute ou si l'on modifie une règle, tout ce qui avait été fait reste valide tant que la situation présente ne concerne pas la nouvelle règle, tout changement est ainsi additif et local. Le système sait différemment tout en sachant plus. Il « assimile » un savoir qui grossit constamment. Dans les programmes procéduraux, en revanche, les interactions étant très étroites, toute modification est difficile et peut se répercuter de façon incontrôlée.

3) Lisibilité :

a) Pour nous, c'est sans doute parce que, comme cela a déjà été souligné, notre propre savoir est modulaire, que les systèmes de production nous semblent plus familiers, plus faciles à lire. Ce point n'est pas à négliger puisque ces systèmes sont d'abord destinés à des médecins, des biologistes ou des architectes par exemple. La « valeur ajoutée » par l'existence effective d'une base de règles est bien réelle dans bien des domaines (Cf. Les systèmes experts d'enseignement assisté par ordinateurs (E.A.O.)).

b) Pour la machine, il est formellement plus simple de vérifier automatiquement la cohérence d'une base de connaissances entrée sous forme de règles plutôt que sous forme de procédures. La contradiction ou la redondance sont en effet aisément détectées par un traitement syntaxique :

$$A \wedge B \rightarrow C \text{ et } A \wedge B \rightarrow \neg C \text{ sont impossibles simultanément ;}$$
$$D \wedge E \wedge F \rightarrow G \text{ entraîne } E \wedge F \rightarrow G \text{ par « subsumption » ;}$$

la première règle est ainsi redondante.

Des incohérences sémantiques peuvent être également relevées, ainsi :

$A \wedge B \rightarrow C$ n'est pas admissible si, d'après les définitions et les

propriétés de A et B , on a nécessairement : $A \rightarrow \neg B$.

4) Faculté d'auto-explication :

Cette propriété est due à la fois aux règles et à leur structure de contrôle externe. Le système remonte facilement la chaîne des règles qu'il a utilisées pour raisonner, ainsi que le jeu de métarègles qui est intervenu.

La représentation homogène de la connaissance, liée au format imposé, autorise la construction des profils et des modèles de règles, qui lui permettent de répondre aux questions et de prévoir une grande partie des corrections à apporter en cas d'erreur dans la base de données (voir paragraphes 7.1.2 et 7.7).

5) Efficacité :

L'expérience montre d'ores et déjà que des systèmes comme MYCIN, PROSPECTOR, ou R1 sont souples et compétents. Les experts eux-mêmes l'affirment. Ces systèmes ne sont plus des jouets de laboratoire, mais des produits opérationnels performants. D'une façon générale, ils s'avèrent suffisamment efficaces et supportent la comparaison avec les systèmes procéduraux comme l'a montré en particulier Rychener (78) sur plusieurs domaines.

Leur structure de contrôle leur permet de prendre en considération de nombreux paramètres caractérisant la situation. Les règles de production tiennent en effet compte étroitement des données particulières du cas traité, ce qui est la raison profonde de leur efficacité.

7.7 La métacognition

D'une façon générale, la métacognition est toute connaissance sur la connaissance. Elle est fondamentale pour des systèmes qui, non seulement utilisent leur base de connaissances telle quelle, mais savent aussi raisonner sur elle, la structurer, l'abstraire, la généraliser et enfin décider des cas où elle peut être utile. Si c'est la tâche des experts d'un domaine de fournir au système la base de connaissances, c'est bien la tâche du système lui-même de gérer celle-ci. Le premier paragraphe traite des objets élémentaires de la métacognition, le second concerne les stratégies et les modes de raisonnement eux-mêmes.

7.7.1 Méta-connaissance des objets de l'univers

Un problème fondé sur une grande masse de connaissances exige en général de manipuler des types de données très complexes et très différents, tant dans leur contenu propre que dans leurs relations avec les autres éléments du système. Il ne suffit pas que l'information qui les concerne figure dans les commentaires des programmes ou dans l'esprit du concepteur, mais bien qu'elle soit accessible au système lui-même. Ainsi, aux objets élémentaires peuvent être liés des concepts ; chacun des concepts, dans MECHO ou MYCIN par exemple, contient plusieurs informations :

- a) Un descriptif qui précise les caractéristiques de la structure de donnée associée ;
- b) Un pointeur vers tous les exemples connus du concept ;
- c) Les relations avec les autres concepts ;
- d) Un pointeur dans la hiérarchie des schémas et vers le groupe de concepts de la même famille.

Lorsqu'une nouvelle règle est énoncée et qu'un mot ou un groupe de mots n'est pas déjà connu du système, il est analysé suivant les différents concepts répertoriés en utilisant le contexte et les mots apparaissant dans la suite de la règle. De plus, si certaines propriétés sont connues pour d'autres exemples du même concept, le système est capable de demander à l'utilisateur de lui préciser convenablement les mêmes propriétés

pour le cas présent. Il détecte également, au besoin, une violation des relations entre concepts.

Ces systèmes ont encore la possibilité de répercuter sur tous les concepts d'un même niveau hiérarchique, une modification imposée par l'expert sur l'un d'entre eux et, aussi, de mesurer quelle est l'importance des changements à effectuer dans la base de données, si l'on ajoute un nouvel exemple à une certaine structure.

De même, les règles de production peuvent être caractérisées par des modèles de règles (décris en [7.2.3](#)). Ces modèles, créés et gérés par le système, rendent compte des régularités rencontrées dans les règles. Eux aussi autorisent les tests de cohérence entre une règle nouvelle et la base ; en outre, ce sont eux qui indiquent à l'expert les tendances principales de l'ensemble des règles, alors même que celui-ci ne les connaît pas. Ce sont eux qui, avec chaque nouvel exemple s'enrichissent quand l'ensemble des règles croît. Dans le système MECHO, c'est par exemple aux énoncés des problèmes eux-mêmes qu'est reliée une forme de méta-connaissance : le système tente systématiquement de reconnaître dans un énoncé une situation standard. Quand il y parvient, il complète alors l'ensemble des faits par toutes les hypothèses physiques, habituellement valides dans le cas reconnu.

7.7.2 Méta-connaissance des stratégies

Dans les systèmes de production, les stratégies elles aussi sont mises sous forme de règles, dites méta-règles puisqu'elles mettent en cause les règles. Elles sont ainsi isolées et le système peut les manipuler. Les heuristiques de guidage de la recherche agissent alors à travers ces méta-règles sous forme qualitative et déclarative, au lieu d'être liées quantitativement à une fonction d'évaluation. Des risques d'erreur sont en effet inhérents aux fonctions numériques d'évaluation : d'abord elles sont peu lisibles, ensuite l'ordre partiel agrégé qu'elles traduisent est d'un calcul complexe : dès lors, toute modification est risquée.

Les stratégies entrées sous forme de méta-règles sont plus lisibles et plus sûres. Les conclusions d'une méta-règle indiquent les actions envisageables dans la situation présente. Elles portent donc sur l'utilité vraisemblable d'un ensemble de règles. Leur effet est double :

- d'une part elles éliminent certaines règles dont les caractéristiques ne sont pas conformes, et l'arbre de recherche est alors élagué ;
- d'autre part, elles opèrent un classement partiel pour les autres règles, et dans l'arbre de recherche, les branches sont partiellement ordonnées.

Bien sûr, il est possible d'augmenter encore le nombre de niveaux dans la connaissance, chaque étage s'empilant sur le précédent et augmentant à mesure l'intelligence que le système a lui-même. On y gagne à chaque fois en généralité, car le même interpréteur peut ainsi travailler dans des domaines divers, et en robustesse, car les modèles supérieurs restent insensibles au changement de la base des connaissances élémentaires.

C'est ainsi que le système CRYSLIS (Engelmore 79) renferme trois degrés de règles distincts. CRYSLIS travaille sur l'analyse de protéines et son espace de recherche est en général trop important du point de vue combinatoire. L'approche retenue classiquement (Cf. paragraphe [7.2.3](#)) ne peut donc convenir. Dès lors, les règles sont d'abord regroupées en sous-ensembles (non disjoints). Chaque sous-ensemble correspond au traitement approprié lorsque les conditions associées sont remplies. La correspondance entre les classes d'éventualités et les sous-ensembles de règles s'effectue ensuite grâce

à des règles de tâches, qui constituent un second niveau de connaissances. Ces règles indiquent donc comment il convient d'accomplir au mieux une tâche donnée.

Le troisième niveau enfin concerne des méta-méta-règles qui définissent les sous-buts à atteindre et les expriment en fonction des règles de tâches.

« Si	deux éléments hypothétiques a et b de la protéine sont déjà placés, avec pour chacun d'eux un coefficient de vraisemblance d'au moins 0,4 ;
ET SI	le nombre d'éléments résiduels dans la séquence ab est au plus 5.
ALORS	activer les règles qui mentionnent la tâche « TRACE-ENTRE-POINTS » »

FIG. 7.23 – Exemple de méta-règle dans CRYSTALIS (Engelmore 79)

Dans l'exemple précédent, le système se reportera ensuite à l'une des règles de tâche qui comportent dans les prémisses la tâche TRACE-ENTRE-POINTS et qui lui indiqueront quelles règles doivent être de préférence utilisées.

Cette façon de regrouper les règles en familles possède un avantage et un inconvénient. L'avantage est d'avoir le contrôle dans le corps même de la règle, chaque règle véhiculant ses propres raisons d'application (Cf. le système AM de Lenat 1977). Une même idée directrice - un plan - peut ainsi se retrouver facilement à travers une famille de règles. L'inconvénient est qu'alors la partie prémisses des règles risque de se trouver considérablement alourdie.

Une alternative pour régler autrement ce problème, vital à la fois pour l'efficacité et pour la lisibilité des systèmes, existe. Elle consiste à autoriser des prémisses de règles de structures de plus en plus complexes. C'est l'objet de la compilation en représentation interne des jeux de règles, décrite au paragraphe 7.5.4.

Le travail de Robert Wilensky (1981) intitulé « Méta-planning » est centré sur la gestion des plans d'action. Indépendamment des domaines, Wilensky propose des métastatégies pour résoudre les problèmes de conflit entre plans, de plans récurrents, de plans concourants. Cette même connaissance est donnée sous forme déclarative à deux programmes différents PAM et PANDORA : l'un fabrique des plans pour résoudre des problèmes, l'autre doit comprendre un récit et donc fabriquer les plans pour les personnages afin d'interpréter leurs actes.

7.8 Conclusion

Les systèmes utilisant les bases de données déclaratives et lisibles par règles de production, relations, prototypes, schémas ou réseaux sémantiques pour représenter et utiliser les connaissances ouvrent véritablement la voie à une nouvelle conception de la programmation : il s'agit ici de transmettre en vrac et le plus simplement possible le savoir de l'homme à un programme expert dans un domaine bien délimité. Un moteur de système expert interprète ainsi un langage déclaratif qui est fondamentalement une structure d'accueil de la connaissance. Une telle approche paraît aujourd'hui indispensable en intelligence artificielle où elle a déjà donné naissance à des systèmes efficaces (paragraphes 7.2 et 7.3).

Les paragraphes 7.4 et 7.5 ont montré que, dans cette conception, les deux propriétés fondamentales étaient, d'une part la totale indépendance des faits élémentaires de

connaissance une fois isolés et mis sous forme convenable et d'autre part, la syntaxe unique et rigide de ces formes. Ces caractéristiques permettent en effet à un système bien conçu de posséder une vision globale de sa connaissance qu'il peut analyser et structurer à nouveau au moyen de (méta)-règles, clauses ou schémas.

Cette simplicité extrême dans la représentation de la connaissance n'implique aucune contrainte désagréable au programmeur ou au spécialiste et semble bien, au contraire, se rapprocher de nos habitudes de pensée les plus naturelles. Elle présente de nombreux avantages et permet en fait de concilier, comme nous l'avons vu aux paragraphes 7.6 et 7.7, des idées présentes dans les travaux actuels à la fois en résolution de problèmes et en compréhension du langage naturel.

D'un point de vue différent, il est évident que le moindre des mérites de ces systèmes experts n'est pas de mettre effectivement au jour, pour nous-mêmes cette fois, des ensembles cohérents de connaissances spécialisées :

- PROSPECTOR peut apprendre des choses à un géologue,
- MYCIN est utilisé dans l'enseignement en médecine,
- PECOS peut donner des idées à tout programmeur.

Enfin ces systèmes mettent en lumière le fait qu'une composante essentielle de notre intelligence est la capacité de gérer une grande quantité d'informations élémentaires. C'est dans cette perspective que trois problèmes fondamentaux, déjà approchés dans les systèmes de production, semblent devoir être particulièrement étudiés dans les recherches en intelligence artificielle pendant longtemps encore : ils ont trait aux actions primitives liées au traitement de toute connaissance :

énoncer - mémoriser - utiliser

- **Enoncer :**

Nous rencontrons continuellement des situations concrètes nouvelles ; notre expérience vient avant tout de notre capacité à abstraire ces situations de façon à être d'abord capable de les décrire à l'aide de symboles généraux. Ces symboles sont souvent ceux du langage courant, mais dans bien des domaines celui-ci ne suffit pas, car il est trop imprécis ou fait de trop nombreuses références à des contextes étrangers à la situation. Chaque spécialiste dégage ainsi peu à peu les bons concepts dans sa discipline et son propre jargon.

Les mauvais livres donnent trop souvent des conseils en décrivant des situations et des actions à entreprendre avec un niveau de détail trop bas : l'effort de généralisation n'y est pas assez grand. L'information s'accumule alors, des résultats sont appris par cœur, isolés des déclencheurs qui en permettraient l'utilisation raisonnée, et toute cette connaissance est inutilisable.

La découverte automatique, par apprentissage, des concepts utiles par rapport aux situations primitives n'est sans doute pas du domaine de l'impossible puisque l'individu doué y parvient quelquefois mais, étant données précisément nos propres difficultés à cet égard, elle ne peut être attendue en intelligence artificielle pour le proche avenir.

- **Mémoriser :**

La mémorisation de toute information fait intervenir des contraintes technologiques

et est liée avant tout à la question de l'efficacité du système d'information. Si, pour des raisons vitales de commodité, les éléments de connaissances sont entrés dans les systèmes de production de façon indépendante comme ils le sont dans un dictionnaire, il est indispensable qu'ils soient stockés de façon organisée. Les règles, regroupées de manière arborescente, constituent alors un réseau dont les noeuds privilégiés sont des modèles qui contiennent la description générale des règles qu'ils commandent.

En outre, il faut que le système sache inverser les règles c'est-à-dire sache déterminer à quelles situations une règle donnée sera utile. Cette opération d'inversion n'est pas simple, elle est liée à la généralisation et elle permet précisément **l'apprentissage de concepts**, descripteurs concis de toutes les situations justiciables d'un même traitement. De façon à éviter le coût élevé qu'entraîne cette inversion, une certaine redondance n'est pas exclue. Il est clair qu'une même information doit être, dans un système intelligent et efficace, présente sous plusieurs formes différentes (Cf. chapitre 9).

- **Utiliser :**

L'accès à l'information pose un problème de reconnaissance de formes, que l'étape précédente avait pour effet de faciliter grâce à la structuration en arbre et à la présence simultanée des formes directes et inverses des règles élémentaires. Le contrôle de cette recherche est encore aidé par des méta-règles et, plus généralement par des modèles de connaissance à des niveaux plus élevés, qui permettent à la fois d'orienter la reconnaissance en fonction d'une stratégie liée au contexte et de retarder le plus longtemps possible les choix.

Plusieurs systèmes experts, fondés sur les règles de production, en particulier CRYSTALIS, ARGOS II, POLITICS, OPS, TANGO, SNARK, ont déjà apporté des éléments de réponses à ces trois questions. Remarquons que c'est avant tout la distinction nette entre la connaissance elle-même et le traitement, par interprétations itératives, qui a permis d'isoler précisément et de pouvoir analyser ces trois problèmes.

Ainsi, les systèmes experts offrent fondamentalement une démarche nouvelle en programmation : des deux principales structures de contrôle habituelles - le test et l'itération - ils ne gardent a priori que la première, qui permet, à elle seule, d'énoncer des faits élémentaires pour rejeter dans un interpréteur, aisément écrit une fois pour toutes, la répétition des cycles de déclenchement de tests et de traitement. Les deux défauts irrémédiables de la programmation classique sont ainsi évités : l'ordre séquentiel du déroulement des opérations et le caractère impératif des instructions élémentaires sont abandonnés au profit de la **donnée en vrac d'une somme d'éléments de savoir isolés où l'on ne préjuge pas à l'avance de l'utilisation qui en sera faite**.

Mais bien plus encore, les systèmes experts permettent finalement d'éviter tout recours à un quelconque langage de programmation. En effet, c'est l'utilisateur, **l'expert lui-même qui définit** ici, à mesure, ses propres besoins, ses propres concepts, **sa terminologie de spécialiste**, en se plaçant d'entrée de jeu au niveau de détail qui lui est familier et qui est adapté à ses problèmes.

A l'inverse, et de façon particulièrement cruciale pour les recherches en intelligence artificielle, les langages de programmation actuels sont toujours de trop bas niveau. Une action simple, un fait de connaissance élémentaire, sont le plus souvent dilués dans un bon nombre d'instructions du code et toute modification est difficile et dangereuse. L'hypothèse de base des chercheurs en intelligence artificielle est donc de considérer le mode de représentation et d'interprétation de connaissances du type « systèmes

experts » comme les langages des années à venir : la modularité absolue, jointe à la lisibilité et à la séparation complète entre données factuelles et contrôle, peut en effet apporter une solution aux problèmes cruciaux de maintenabilité du savoir, à sa communication entre hommes et entre programmes, ainsi qu'à la spécification et à l'écriture de gros logiciels non nécessairement algorithmiques.

Pour faire de ces systèmes des outils fiables et utilisables hors des laboratoires, il reste certes plusieurs travaux à mener à bien. Deux tâches en particulier doivent encore être développées avec rigueur. Il s'agit, en premier lieu, de l'étude formelle des langages d'expression des règles et de métarègles, de recherche sur leur syntaxe et leur sémantique, des types de logiques sous-jacentes, des primitives fondamentales au raisonnement : il s'agit de faire un travail analogue à celui entrepris sur les langages de programmation, mais dans lequel il faut s'affranchir des interpréteurs et des ordinateurs actuels.

En second lieu, il reste à sortir les recherches en intelligence artificielle des laboratoires universitaires. En particulier, pour les systèmes experts, la construction d'interpréteurs généraux, efficaces, documentés et dotés de mécanismes d'inférences assez riches n'est pas chose acquise. De tels moteurs doivent maintenant pouvoir être remis, comme le sont les compilateurs, en toute confiance aux nouveaux utilisateurs.

Les systèmes existants ont prouvé la faisabilité d'une approche radicalement nouvelle en informatique : la formalisation de notre savoir en unités déclaratives indépendantes, de structure bien définie - règles de production, frames, scripts ou réseaux sémantiques - qui apporte une solution élégante, d'utilisation aisée et de conception simple, aux problèmes de représentation de la connaissance et évite les difficultés classiquement liées aux langages de programmation, en **permettant à tout un chacun de s'adresser à de tels systèmes avec ses propres mots**.

7.9 Votre moteur de système expert en kit

Nous nous limitons ici au cas où tout fait de la base de faits BF est de la forme :

$$aRb$$

où a et b sont deux constantes (mots arbitraires de l'univers) : a et b , qui sont a priori indépendants, sont liés dans BF par la relation R (R peut être, par exemple : « est », « appartient à », « égale », « contient », « vaut », ... au gré de l'expert). Le fait aRb correspond alors à une proposition logique habituelle, qui est interprétée comme vraie ou fausse.

Exemples :

L'anémone est une-fleur-dicotylédone
ou : L'organisme a-le-type-Gram-positif.

De la même façon, toute règle de BR a la forme : $\wedge p_i \rightarrow \wedge q_j$

dans laquelle les prémisses p_i et les conséquents q_j sont des propositions sous forme de triplets (mot, relation, mot). Nous nous restreignons donc ici à un univers sans variables : logique des propositions ou niveau zéro. Partant des faits initiaux et des règles données, le *moteur zéro* a pour but d'inférer de nouveaux faits en déclenchant toute règle la base de règles BR dont les prémisses sont satisfaites. Les conséquences de la règle sont alors simplement ajoutées à la base de faits BF.

Comme indiqué dans le corps de l'article à la section [7.3](#), le noyau du moteur est constitué du cycle de base :

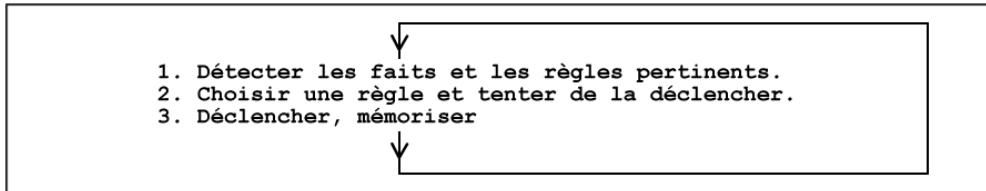


FIG. 7.24 – Cycle de base d'un système expert

Deux stratégies générales sont ici possibles :

- Le chaînage avant : on part des faits connus donnés par l'expert ; on déclenche toute règle dont les prémisses sont satisfaites : on ajoute les faits conséquents ; on recommence jusqu'à « saturation » du système (plus de nouvelles déductions).
- Le chaînage arrière : supposant que l'on cherche une réponse à une question de l'expert et que le nombre de réponses possibles soit fini, on considère la première d'entre elles : c'est le but.

On regarde alors toutes les règles qui ont le but dans leurs conséquents. S'il n'y en a pas, on passe au but suivant. Sinon chacune de ces règles est considérée : si toutes ses prémisses sont satisfaites dans BF, le but est atteint, sinon on enregistre les prémisses inconnues comme autant de nouveaux buts et on recommence le cycle. On remonte ainsi du but aux faits initiaux. Le système aura en outre le droit de poser des questions à l'expert à chaque fois qu'il en aura besoin, au cours de raisonnement, d'un fait non donné et non déductible par règle.

La procédure générale est ici, par construction, récursive, puisque la vérification d'un fait se fait par le test des règles qui concluent à ce fait, et que ce test lui-même s'effectue par vérification des prémisses des règles.

Construisons donc une fonction récursive VERIFIER appelée par le programme principal sous la forme :

$$\text{Résultat} = \text{VERIFIER}(b, BF, BR)$$

où b est le but cherché, BF la base de faits initiale, BR la base de règles. La syntaxe d'une règle est ici simplement :

$$\begin{aligned}
 <\text{Règle}> &= \text{SI } <\text{prémisses}> \text{ ALORS } <\text{conséquent}> \\
 <\text{Prémisses}> &= <\text{fait}> / <\text{fait}> <\text{prémisses}> \\
 <\text{Fait}> &= <\text{mot}> / <\text{mot}> \\
 <\text{Conséquent}> &= <\text{prémisses}>
 \end{aligned}$$

Fonction VERIFIER renvoie BOOLEEN

<u>Constantes globales</u>	base de règles :	<u>ensemble de règles</u>
	demandable :	<u>ensemble de PROPOSITIONS</u>
but, base de faits :	<u>ensemble de PROPOSITIONS</u>	

Données

$\left\{ \begin{array}{l} \text{Le but est déduit récursivement de la base de faits et de la base de règles} \\ \text{avec questions éventuelles à l'expert.} \end{array} \right\}$

variable ok : BOOLEEN ; ok : \leftarrow faux

Si $b \in BF$ ALORS ok \leftarrow vrai

{ b déjà établi }

{ premier cas de succès trivial }

FSI

{ Sinon on recherche s'il est déductible }

REPETER POUR toute $r \in BR$ avec $b \in \text{conséquents}(r)$ TANT QUE ok = faux :

ok = PROUVER (Prémisses(r))

{ Appel à la fonction PROUVER qui vérifie que toutes les prémisses de r sont vraies. }

C'est le deuxième cas possible de succès. }

FR

{ sinon voir si b est demandable }

SI ok = faux et si $b \in$ demandable

| ALORS poser la question ; ok \leftarrow réponse(b)

FSI

{ dans tous les cas de succès, mémoriser dans BF }

SI ok := vrai ALORS BF \leftarrow BF \cup { b }

FSI

Renvoyer (b, ok)

FIN VERIFIER

Fonction PROUVER renvoie BOOLEEN

{ Mêmes constantes globales }

Données : B ensemble de buts

ok \leftarrow vrai

REPETER POUR TOUT $b \in B$ et TANT QUE ok = vrai

ok \leftarrow VERIFIER (b)

FR

{ il faut que tous les buts de B soient prouvés }

Renvoyer ok

FIN PROUVER

FIG. 7.25 – Un moteur essentiel en logique des propositions

VERIFIER trouve la valeur de vérité de b par chaînage arrière. Le résultat est booléen (vrai ou faux) suivant que b est finalement prouvée ou non. VERIFIER appelle elle-même la procédure PROUVER qui vérifie quant à elle qu'un ensemble de buts, cette fois, est vérifié. PROUVER fait pour cela appel à la fonction VERIFIER.

Les objets manipulés dans VERIFIER et PROUVER sont de trois types : PROPOSITION, ensemble-de PROPOSITION et REGLE. A tout objet r de type REGLE

correspond deux attributs : PREMISSES (r) et CONSEQUENTS (r) qui sont l'un et l'autre de type ensemble-de PROPOSITION.

Enfin certaines propositions appartiennent a priori à l'ensemble DEMANDABLE : elles peuvent faire, en dernier ressort, l'objet d'une question directe à l'expert. La fonction QUESTION (p) pose la question p et renvoie vrai ou faux selon la réponse de l'expert. Il y a ainsi finalement trois façons de montrer qu'une proposition p a la valeur vraie :

- par examen direct de BF,
- par déduction récursive à l'aide de BR,
- par question directe quand p est demandable.

La fonction VERIFIER s'exprime comme suite, où les notations ensemblistes (appartenance) et les primitives de haut niveau (Meyer 78) permettent une présentation claire et concise.

A vous de programmer ceci sur votre micro-ordinateur personnel et de faire, par exemple, tourner ce « moteur zéro » de système expert sur le jeu de règles suivant (les propositions aRb sont ici de simples mots : « la plante possède une fleur » est abrégé en : « fleur ») :

a)	SI	fleur \wedge graine	ALORS	phanérogame
b)	SI	phanérogame \wedge graine nue	ALORS	sapin
c)	SI	phanérogame \wedge 1cotylédone	ALORS	monocotylédone
d)	SI	phanérogame \wedge 2cotylédone	ALORS	dicotylédone
e)	SI	monocotylédone \wedge rhizome	ALORS	muguet
f)	SI	dicotylédone	ALORS	anémone
g)	SI	monocotylédone \wedge \neg rhizome	ALORS	lys
h)	SI	feuille \wedge \neg fleur	ALORS	cryptogame
i)	SI	cryptogame \wedge \neg racine	ALORS	mousse
j)	SI	cryptogame \wedge racine	ALORS	fougère
k)	SI	\neg feuilles \wedge plante	ALORS	thallophyte
l)	SI	thallophyte \wedge chlorophylle	ALORS	algue
m)	SI	thallophyte \wedge \neg chlorophylle	ALORS	champignon
n)	SI	\neg feuille \wedge \neg fleur \wedge \neg plante	ALORS	colibacille

FIG. 7.26 – Jeu simplifié de règles pour le moteur zéro

Si la base de faits initiale est : {rhizome, fleur, 1cotylédone, graine}, la fonction VERIFIER donnera, avec but = champignon ; QUESTION : thallophyte ? (cas 3 ; on ne peut déduire thallophyte, il faut le demander). Si, par exemple, la réponse expert est : (faux) VERIFIER renvoie faux.

Si but = muguet, VERIFIER appelle PROUVER pour tester les prémisses de la seule règle -e- qui permet de conclure. On remonte donc à VERIFIER pour prouver le premier sous-but : monocotylédone, qui ne peut venir que de la règle c). PROUVER est à nouveau lancée sur {phanérogame et 1cotylédone}. Le premier but passe par la règle a), dont les deux prémisses sont déjà dans la base de faits (premier cas de succès), celle-ci s'enrichit donc de phanérogame. Puis, comme 1cotylédone est connu, on ajoute encore monocotylédone. Remontant la pile de récursion, on revient sur la seconde prémissse de la règle e) qui est connue et le système répond finalement vrai.

On notera que c'est la réalisation précise de l'opération « b appartenant à buts » qui déterminera finalement le mode d'exploration de l'arbre des propositions potentielles : si buts est géré comme une pile, on obtiendra une exploration « en profondeur d'abord »

(voir par exemple la version BASIC, de quelques 400 lignes pour micro-ordinateur dans BYTE septembre 1981 ou FARRENY 1985).

L'efficacité de la fonction récursive VERIFIER peut être immédiatement améliorée par transformation des deux paramètres, buts et base de faits, en variables globales, avec gestion explicite dans le programme.

Enfin, pour ne pas répéter des tâches inutiles, il faut distinguer les faits faux et les faits inconnus. Il est possible de tester en particulier des conditions négatives dans les règles en les faisant correspondre aux faits connus comme faux. En outre, il n'est pas difficile de faire fonctionner également ce « moteur zéro » en chaînage avant. L'arbre de recherche est alors parcouru par activation des règles en fonction des conclusions obtenues jusqu'à « saturation » (plus de déduction possible) de la base de faits.



Chapitre 8

ALICE

« La différence fondamentale n'est pas entre le naturel et l'artificiel, mais entre les systèmes programmables et ceux qui ne le sont pas. »

Herbert A. Simon (1977)

ALICE est le nom d'un système informatique qui comprend à la fois : **un langage de haut niveau** permettant de poser des problèmes ; **un module général résolution de problèmes** utilisant des méthodes d'intelligence artificielle.

La conception d'ALICE a eu pour point de départ deux idées :

A. La programmation classique est coûteuse

En effet, pour chaque nouvelle application, les informaticiens écrivent un nouveau programme. En particulier, toute modification dans les spécifications du cahier des charges conduit à un coût de reprogrammation très élevé : 75 \$ l'instruction initiale, mais 4000 \$ l'instruction corrigée (Department of Defense, USA, 1972). Ceci pour deux raisons :

- a) Les effets de bords sont mal contrôlés car, dans les langages procéduraux actuels, toute correction est susceptible d'avoir une portée imprévue, même si la méthodologie (programmation structurée, programmation modulaire, types abstraits) est bonne et procure quelques garde-fous.
- b) Un changement de spécifications risque, en fait, de changer radicalement les structures de données et l'algorithme de résolution. Ainsi, si les livres fournissent des exemples académiques, toute application éloigne du cas idéal, et tout est continuellement à repenser et à reprogrammer.

B. Tous les langages de programmation actuels sont des langages impératifs.

Ceci signifie qu'il faut, à chaque fois, dire à la machine, pas à pas. ce qu'elle doit faire. Or, après tout, seul importe à l'utilisateur l'énoncé du problème et son résultat. Spécifier continuellement toutes les étapes intermédiaires n'est, à terme, guère supportable. Autrement dit, il convient d'envisager un traitement informatique **où l'énoncé d'un problème est complètement séparé de sa résolution**. On peut dès lors songer à une solution idéale dans laquelle l'utilisateur, de manière conversationnelle, soumet *un* énoncé dans un langage descriptif proche de la langue naturelle. L'ordinateur donne, **en temps réel**, une solution que l'utilisateur critique en corrigeant éventuellement quelques spécifications du premier énoncé pour un nouveau passage.

Le temps de réponse total est ainsi réduit de quelques semaines (ou quelques mois) à quelques minutes. Telle est l'approche proposée dans le système ALICE.

En réalité, les arguments 1 et 2 se rejoignent pour définir : **un langage descriptif** (déclaratif) pour poser un problème quelconque de manière purement formelle ; **un système de résolution** suffisamment général pour pouvoir interpréter toute requête de l'utilisateur et être capable de prendre en compte toutes les particularités du problème traité, à la fois dans sa définition formelle, dans ses ordres de grandeurs, et dans les ressources disponibles. Le premier point correspond aux travaux actuels sur les langages de spécifications (Abrial, Guttag, Liskov, Meyer). Le second point concerne la méthodologie de la programmation et la résolution de problèmes en général (Arsac, Nilsson, Pair, Simon). Le système ALICE sait interpréter la spécification d'un problème jusqu'à sa résolution complète. En outre, nous décrirons au paragraphe 9 comment ALICE peut actuellement, dans certains cas, résoudre un problème directement à partir de l'énoncé en français courant.

- *Note*

Il a été jugé préférable de diviser la description du logiciel ALICE en deux parties : la première, constituée des paragraphes 1 et 2. présente la famille de problèmes que l'on peut poser au système, le langage d'entrée, une description assez brève et générale des méthodes de résolution utilisées, et la résolution de trois exemples. Les paragraphes 3 à 5 constituent la deuxième partie. Les procédures, les critères, les représentations utilisées, la gestion des choix et la manipulation des contraintes y sont donnés selon un point de vue plus technique. Une discussion de l'ensemble des résultats fait l'objet du paragraphe 8. Enfin une recherche sur l'acquisition des énoncés en français est présenté en 9.

« **L'homme est le spécialiste de la non-spécialisation.** »

Konrad Lorenz (1950)

Le tout et la partie dans la société animale et humaine.

8.1 Description générale du système

Il s'agit de concevoir un système qui d'une part, admette des énoncés purement déclaratifs, ce qui signifie qu'aucun algorithme, instruction ou conseil n'a besoin d'être donné, et qui d'autre part, soit à même de résoudre une classe suffisamment générale de problèmes, éventuellement difficiles et de taille importante.

Le langage d'énoncés de problèmes doit permettre seulement de **spécifier** l'application à traiter. Il a pour base essentielle le langage mathématique habituel.

Pour le système de résolution, un choix était possible : soit compiler brutalement l'énoncé de départ et le traduire en un algorithme spécifique, adapté au problème posé ; soit interpréter intelligemment **l'énoncé avec ses données** à l'aide d'un module général. La seconde solution, dans la mesure où il est possible de la programmer, se révèle largement la plus efficace : comme nous le faisons nous-mêmes, le système peut alors, en effet, s'adapter continuellement au cas traité en tenant compte de toutes ses particularités. **Il ne déroule pas un algorithme unique et prédéfini, mais emploie à chaque pas la procédure adéquate.**

Le résultat surprenant est qu'alors, en plusieurs occasions, sur des problèmes difficiles, ce programme général est plus performant que les meilleurs programmes particuliers

actuellement disponibles. Ce point est notamment expliqué et discuté au paragraphe 8. Plus d'une centaine de problèmes différents et de natures très diverses ont, à ce jour, été effectivement résolus par le système. (Cf. paragraphe 8.7 : table des résultats).

8.2 Le langage ALICE

ALICE = A Langage for Intelligent Combinatorial Exploration.

L'ossature générale du langage est donnée par le vocabulaire de la théorie des ensembles et de la logique classique : plusieurs compléments commodes de la théorie des graphes, en particulier les notions de chemin et de successeur, ont été ajoutées. L'utilisation d'ensembles, de fonctions, de produits cartésiens, de matrices, permet de décrire les objets fondamentaux de l'application concernée. Les relations qui lient entre eux ces objets peuvent être précisées grâce aux opérateurs usuels tels que :

$\in, \forall, \exists, \Rightarrow, \Leftrightarrow, \text{ET}, \text{OU}, \neg, =, \neq, \geq, \leq, +, -, *, /, \sum,$ modulo.

La classe des problèmes acceptés par ALICE est donc celle des énoncés de forme générale :

Trouver $x, x \in X, X$ fini donné, qui vérifie un ensemble de contraintes $K(x).$

Cette classe est très générale. On y trouve tous les problèmes d'algorithmique classique (tris, mises à jour, comptabilité, gestion, résolution de systèmes algébriques, ...), les problèmes de recherche opérationnelle et d'optimisation (voyageur de commerce, recouvrement, knapsack, localisation, planning, ordonnancement), mais aussi beaucoup d'autres moins purs, moins classiques. La seule limitation est que le vecteur d'inconnues x est astreint à appartenir à un ensemble X fini.

L'aspect mathématique des énoncés souvent difficiles donnés ici ne doit pas masquer le fait que des problèmes plus simples sont plus aisément posés et résolus.

Énoncés en langage ALICE.

L'énoncé d'un problème se compose de quatre parties : la partie définition, commandée par le mot-clé « *soit* », décrit les objets sur lesquels portent le problème et donne leurs types ; la partie but spécifie le but à atteindre, les quantités inconnues à déterminer - cette partie est introduite par le mot-clé « *trouver* » ; la partie contrainte commence par le mot « *avec* » ; elle précise complètement les autres caractéristiques du problème et les relations que doivent mutuellement satisfaire toutes les quantités qui interviennent. L'optimisation d'une fonction, un coût par exemple, peut être imposée dans, cette partie : les données numériques ou formelles sont finalement fournies, en accord avec les déclarations de la première partie, pour définir intégralement le problème posé.

- *Exemple 1 :*

« *Dans un ordinateur multiprocesseur, N tâches, dont on suppose les durées connues, sont prêtes à être lancées. Elles peuvent l'être indifféremment sur chaque processeur, mais on impose qu'elles soient toutes achevées au plus tard à une date limite L donnée. En outre, on veut minimiser le nombre de processeurs effectivement utilisés. Etablir l'ordonnancement optimal des tâches.*

Ce texte français est traduit en ALICE de la manière suivante : Définissons l'ensemble T des tâches, que nous conviendrons de numérotter de 1 à N . Il s'agit de trouver pour chaque tâche t de T le processeur sur lequel t doit être exécutée ; soit donc $p(t)$ le numéro du processeur sur lequel est exécutée la tâche t . On veut que, sur l'ensemble

des tâches, $p(t)$ soit minimal, tout en respectant L , la date limite imposée. Autrement dit, on veut que $\max p(t)$, pour t élément de T , soit minimal avec la contrainte suivante sur la date limite : si $\text{durée}(t)$ est la durée de la tâche t

$$\sum \text{durée}(t) \leq L$$

la somme étant prise pour toutes les tâches passant sur un même processeur, c'est-à-dire, si r est le numéro du processeur, sur toutes les tâches t telles que $p(t) = r$. L'énoncé définitif en ALICE sera donc :

SOIT		
CONSTANTES N, L	Nombre de tâches, date limite,	
ENSEMBLE $T = (1, N)$	Tâches numérotées de 1 à N ,	
COEFFICIENT durée sur T	N, L , durée seront lues,	
TROUVER FONCTION $p : T \rightarrow T$	a priori N processeurs,	
	peuvent être nécessaires	
AVEC		
$\forall r, r \in t$	(pour tout processeur)	
$\sum_{t \in T} \text{durée}(t) \leq L$		
$p(t) = r$		
AVEC		
$\text{MIN } \underset{p}{\text{MAX}} \underset{t \in T}{p(t)}$	(Le nombre des processeurs utilisés doit être minimisé)	
FIN		

Ceci définit l'énoncé formel du problème. Alice travaille, on l'a vu, avec les données numériques. Celles-ci sont lues dans une quatrième partie, suivant l'ordre même où elles ont été introduites dans le texte : N et L , puis le vecteur durée. Dans les applications traitées par le système, le nombre de tâches était de plusieurs centaines (Cf. les résultats : paragraphe 7) ; nous donnons ici un exemple comportant seulement neuf tâches, de durées respectives 6, 2, 8, 3, 6, 15, 6, 9 et 13 avec $L = 16$, afin de pouvoir suivre, au paragraphe 3, la résolution complète du problème par le système. Les trois dernières lignes de l'énoncé sont alors :

$$\begin{array}{ll} 9, 18 & (\text{Valeurs de } N \text{ et } L) \\ 6, 2, 8, 3, 6, 15, 6, 9, 13 & (\text{durées}) \\ \text{FIN} & \end{array}$$

On notera que la résolution serait radicalement différente si les durées étaient différentes, en particulier si la durée de la tâche 1 passait de 6 à 7 : en effet, on verra que, loin d'établir une méthode définie pour le problème formel, le système s'adapte étroitement aux données (Cf. paragraphe 3).

- *Exemple 2 :*

Il existe des programmes du type « Analyse numérique » pour résoudre des problèmes de programmation mathématique, linéaire ou non linéaire, comme : « Trouver toutes les solutions en variables booléennes du système :

$$\left\{ \begin{array}{lcl} 0 & \leq & x_1 x_4 + 3x_2 x_5 - x_1 - 2x_3 - x_4 x_5 - 1 \\ 0 & \leq & 2x_1 + 5x_3 - 2x_4 - 3x_5 - 2 \\ 0 & \leq & x_2 x_4 + x_1 x_4 + 2 - 5x_1 \end{array} \right. \Rightarrow$$

Mais nous verrons, au paragraphe 3, toute la puissance apportée dans ce genre d'énoncé

par la démarche non-algorithmique d'ALICE et l'efficacité de la **manipulation symbolique des expressions**. L'énoncé est ici simplement :

SOIT

ENSEMBLE

$$I = \llbracket 1, 5 \rrbracket$$

ENSEMBLE

$$B = \llbracket 0, 1 \rrbracket$$

TROUVER FONCTION $x : I \rightarrow B$

$$\begin{aligned} \text{AVEC } 0 &\leq x(1)x(4) + 3x(2)x(5) - x(1) - 2x(3) - x(4)x(5) - 1 \\ 0 &\leq 2x(1) + 5x(3) - 2x(4) - 3x(5) - 2 \\ 0 &\leq x(2)x(4) + x(1)x(4) + 2 - 5x(1) \end{aligned}$$

FIN

Il n'y a pas de données numériques.

- *Exemple 3 :*

On cherche à définir, dans une agglomération urbaine, les meilleurs emplacements des centres de secours d'urgence (ambulances, pompiers, ...). Pour chaque emplacement possible, la région couverte dans un délai estimé raisonnable est donnée par le schéma 8.1. Toutes les régions doivent finalement être couvertes. En outre, un coût de construction et d'utilisation de chaque emplacement est donné et il s'agit de minimiser le coût total. (Ce problème se rencontre aussi sans fonction à optimiser).

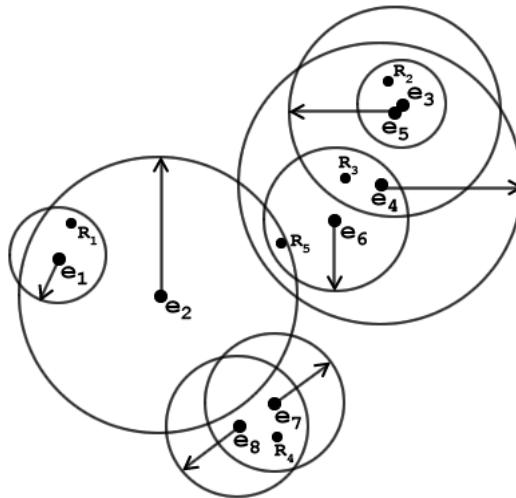


FIG. 8.1 – Localisation des centres de secours

e_1 à e_8 = emplacements possibles ; soit $E = \{e_1, \dots, e_8\}$.

Les cercles indiquent alors les zones couvertes, r_1 à r_5 = régions à desservir, soit $R = (r_1, \dots, r_5)$.

Vecteur de coûts C_1 à C_8 associé aux emplacements = (3, 7, 5, 8, 10, 4, 6, 9).

Il faut ici trouver une fonction f qui, à chaque région, associe au moins un centre, soit $f : R \rightarrow E$. Mais de plus, seul un sous-ensemble de E , donné par la figure 8.1, convient pour chacune des régions. Le langage ALICE permet ici de disposer, dans la déclaration de f , de l'option « successeur » qui traduit cette restriction. Les successeurs de r_5 (c'est-à-dire les images possibles de r_5 par la fonction f) sont par exemple : e_2, e_4, e_6 . Moyennant cette abréviation, l'énoncé du problème devient :

SOIT	CONSTANTES : N_R, N_E	nombre de régions de d'emplacements
	ENSEMBLE : $R = \llbracket 1, N_R \rrbracket$	
	$E = \llbracket 1, N_E \rrbracket$	
	COEFFICIENT C sur E	(coûts des emplacements)
TROUVER	FONCTION $F : R \rightarrow E$	SUCCESEUR
AVEC	$\forall r, r \in R$	
	$f(r) = e \Rightarrow \forall s, s \in R, e = \text{SUCCESEUR}(s), f(s) = e$	
		(si la région s a pour successeur un emplacement e déjà retenu, alors elle est aussi couverte par e)
	MIN $\sum_{e \in E} C(e)$	(minimisation des coûts retenus)
	$\forall r, r \in R$	
	$f(r) = e$	
FIN		
	5, 8	(valeurs de N_R et de N_E)
	3, 7, 5, 8, 10, 4, 6, 9	(vecteur coût)
	1, 2	} Liste des successeurs
	3, 4, 5	
	4, 5, 6	
	7, 8	
	2, 4, 6	
FIN		

D'autres énoncés équivalents seraient, ici, également possibles : une formulation en variables bivalentes, comme dans l'exemple 2, conviendrait tout à fait : elle serait néanmoins plus lourde.

La résolution de tous ces exercices se fait « en temps réel » et la réponse est quasi immédiate même quand, sur des problèmes issus de cas réels, la taille des données est plus importante. La résolution fournie par ALICE pour ces trois cas est donnée au paragraphe 3.2.3.

D'autres mots clés facilitent de la même façon l'énoncé d'autres contraintes classiques. Le système connaît ainsi également les concepts de surjection, injection, bijection. Il comprend tous les symboles logiques habituels, il sait ce qu'est un arbre, un chemin, un circuit. ALICE a été conçu pour que les énoncés restent concis, les plus compréhensibles possible, et proches de leur formulation en langage courant. De manière plus précise, la table 8.1 donne le dictionnaire complet des mots connus d'ALICE et leur signification.

(Le rang d'un symbole est le nombre de termes qu'il commande.)

Tout symbole n'appartenant pas au dictionnaire est considéré comme le nom d'un nouvel objet qui doit être déclaré par une définition SOIT.

Les règles syntaxiques de formation des phrases du langage sont données sous forme normalisée de Backus par la table 8.2. La forme polonaise préfixée, qui est non-ambiguë et permet d'écrire les contraintes sur une seule ligne et sans parenthèse, a été préférée à la notation usuelle. Nous garderons cependant cette dernière, par commodité, dans ce texte.

- *Notation :*

Le symbole $< x >^*$ représente l'entité x concaténée avec elle-même, un nombre quelconque de fois.

| représente une alternative.

SYMBOLES	SIGNIFICATION	COMMENTAIRES	RANG
SOIT	Définition (partie 1)	Données éventuelles	2
TROUVER	But (partie 2)		1
AVEC	Contrainte (partie 3)		1
FIN	fin de l'énoncé formel		0
CST	constante		1
ENS	ensemble		1
COE	vecteur de coefficients numériques		1
COA	vecteur de coefficients alphanumériques		1
MAT	matrice		2
FON	fonction		2
INJ	injection		2
SUJ	surjection		2
BIJ	bijection		2
SUC	successeurs imposés	Suivis d'une liste d'objets (Cf. texte)	0
PRE	prédécesseurs	idem	0
SYM	symétrie des successeurs		0
DIS	disjonctions imposées	i et j en disjonction : $i \neq j \Rightarrow f(i) \neq f(j)$	0
VAL	coût associé à chaque image		0
DMA	{ degrés maximum et minimum dans l'espace image		0
DMI	{ mum dans l'espace image		0
CHE	chemin		2
ARB	arbre		0
CIR	circuit		0
MIN	minimum		2
MAX	maximum		0
$>, <, \leq, \geq, \neq, =, +, -, *, /$		sens habituel	0
MOD	modulo		2
OBJETS	définition d'un ensemble par ses éléments		n
SUR	lien		
	coeffcient/ensemble		1
PDT	produit catésien		2
INC	inclus dans		1
APP	appartenant		2
\rightarrow	de ... vers	définition d'une fonction	1
QQS	quel que soit		2
EXI	il existe		2
NON	négation logique		1
UN	union		2
IN	intersection		2
ET, OU, $\Rightarrow, \Leftrightarrow$		sens habituel	2
SIG	sigma	sens habituel	2

TAB. 8.1 – Dictionnaire du langage ALICE

$\langle \text{énoncé} \rangle$	=	$\langle \text{phrase} \rangle^* \langle \text{données} \rangle$
$\langle \text{phrase} \rangle$	=	$\langle \text{déclaration} \rangle \mid \langle \text{contrainte} \rangle \mid \text{FIN}$
$\langle \text{déclaration} \rangle$	=	SOIT CST $\langle \text{nom constante} \rangle \mid$ SOIT ENS $\langle \text{définition ensemble} \rangle \mid$ SOIT vecteur $\langle \text{nom vecteur} \rangle \text{ SUR } \langle \text{nom} \rangle \mid$ SOIT MAT $\langle \text{nom matrice} \rangle \text{ PDT } \langle \text{nom} \rangle \langle \text{nom} \rangle$ TROUVER $\langle \text{fonction} \rangle \langle \text{nom fonction} \rangle : \langle \text{nom} \rangle \rightarrow \langle \text{nom} \rangle$ $\langle \text{option} \rangle^*$
$\langle \text{définition ensemble} \rangle$	=	[$\langle \text{nom constante} \rangle, \langle \text{nom constante} \rangle \rangle \mid$ OBJET $\langle \text{nom objet} \rangle^* \mid$ UN $\langle \text{nom} \rangle \langle \text{nom} \rangle$ IN $\langle \text{nom} \rangle \langle \text{nom} \rangle$
$\langle \text{vecteur} \rangle$	=	COE COA
$\langle \text{fonction} \rangle$	=	FON INJ SUR BIJ
$\langle \text{option} \rangle$	=	SUC DIS DMI DMA CIR ARB SYM VAL
$\langle \text{contrainte} \rangle$	=	AVEC $\langle \text{expression logique} \rangle$
$\langle \text{expression logique} \rangle$	=	$\langle \text{opérateur binaire} \rangle \langle \text{expression logique} \rangle$ $\langle \text{expression logique} \rangle \mid$ NON $\langle \text{expression logique} \rangle \mid$ $\langle \text{opérateur algébrique} \rangle \langle \text{expression expression} \rangle \mid$ MIN $\langle \text{expression} \rangle \mid$ MAX $\langle \text{expression} \rangle \mid$ $\langle \text{quantificateur} \rangle \text{ APP } \langle \text{nom} \rangle \langle \text{nom} \rangle \langle \text{expression logique} \rangle$
$\langle \text{opérateur binaire} \rangle$	=	$\Rightarrow \mid \Leftrightarrow \mid \text{OU} \mid \text{ET}$
$\langle \text{quantificateur} \rangle$	=	QQS EX
$\langle \text{opérateur algébrique} \rangle$	=	$< \mid > \mid \geq \mid \leq \mid = \mid \neq$
$\langle \text{expression} \rangle$	=	$\langle \text{opération} \rangle \langle \text{expression} \rangle \langle \text{expression} \rangle \mid$ $\langle \text{nombre} \rangle \mid \langle \text{nom} \rangle \mid$ SIG APP $\langle \text{nom} \rangle \langle \text{nom} \rangle \langle \text{expression} \rangle$
$\langle \text{opération} \rangle$	=	$+ \mid - \mid * \mid / \mid \text{MOD}$
$\langle \text{nombre} \rangle$	=	$\langle \text{« chiffre »} \rangle \mid \langle \text{nombre} \rangle \langle \text{« chiffre »} \rangle$
$\langle \text{nom} \rangle$	=	$\langle \text{« lettre »} \rangle \mid \langle \text{nom} \rangle \langle \text{« lettre »} \rangle$

TAB. 8.2 – Syntaxe du langage ALICE

8.3 Le module de résolution de problème

Nous donnons ici une description globale de ce module. Sa description détaillée fait l'objet du paragraphe 5. ALICE imite la démarche de l'homme quand il s'attaque à un problème :

- **Le problème est continuellement interprété** : le sous-problème le plus simple, en fonction d'une mesure de complexité continuellement remise à jour, est résolu à chaque étape. **L'ordre de résolution se fait à la volée**, suivant l'état courant des contraintes et des valeurs numériques. Loin de suivre, comme en algorithmique classique, un ordre figé des instructions, arbitrairement décidé par le programmeur. ALICE choisit à chaque instant la difficulté à laquelle elle va s'attaquer. Cette conduite est plus proche de la nôtre : nous n'avons pas, par exemple, un algorithme de tri unique, mais nous savons choisir en fonction des données particulières une méthode précise, et même en **changer en cours de route**.
- **Une représentation adaptée** est prévue pour de nombreux types de contraintes.

Toutes les contraintes simples, c'est-à-dire mettant en cause au plus deux objets, sont directement intégrées dans la représentation interne à l'aide d'un *hypergraphe* (Cf. paragraphe 5. 1). Les autres contraintes sont traitées sous une forme équivalente à la notation algébrique habituelle.

- **Un plan de résolution** est construit. Il s'appuie sur une analyse et une interprétation approfondie de la situation initiale. Il dépend notamment de la difficulté estimée du problème. Les variables et les contraintes importantes sont en particulier repérées ; elles serviront à établir les **critères** qui décideront de l'ordre de résolution du problème.

Trois actions principales sont entreprises par le système lors de sa résolution :

- **Propager les contraintes :**

Il s'agit de combiner les informations disponibles de façon à en déduire le plus grand nombre de nouveaux renseignements. Cette analyse se fait du plus simple au plus compliqué : plus courte est l'expression d'une contrainte, plus celle-ci a de chance d'être aisément utilisable. Cette propagation se fait par pointeurs dans la représentation graphique interne pour les contraintes à deux variables ; elle se fait formellement sur la notation algébrique pour les autres. ALICE reste le plus longtemps possible dans cette phase de propagation des contraintes. Cependant, pour beaucoup de problèmes (tous les problèmes NP-complets, voir chap. ??). ceci ne suffit pas pour conclure. Un autre type d'action consiste à :

- **Faire des hypothèses :**

ALICE a donc **retardé les choix aussi longtemps qu'il était possible** de le faire. S'il s'avère qu'il est impossible de résoudre complètement ainsi le problème. Il lui faut tenter de faire un choix intelligent. Au lieu de se lancer dans une énumération aveugle et standardisée, ALICE va fixer la valeur la plus probable pour l'élément le plus intéressant (Cf. infra 5.3). Ceci fait. ALICE retourne, bien entendu, à la phase d'analyse et de propagation de contraintes. Il reste, finalement, dans certains problèmes, à trouver la meilleure solution au sens d'une fonction de coût donnée. Il faut en particulier :

- **Atteindre et démontrer l'optimum :**

Outre la construction de critères qui tendent à améliorer le coût de chaque choix, ALICE connaît dès méthodes d'agrégation et de relaxation de contraintes qui permettent de bâtir un encadrement de la valeur de l'optimum et de **démontrer** finalement celui-ci, en revenant une nouvelle fois à la phase de propagation.

8.3.1 La propagation des contraintes

Cette phase est principalement basée sur l'algorithme d'unification (Cf. chapitre 3). Cet algorithme permet essentiellement d'effectuer un calcul formel sur les expressions, tout comme nous le faisons à la main. Il permet en particulier de faire des simplifications :

$$\begin{array}{ll} 4 \leq 3 & \text{devient : Faux} \\ 2x - 3y + x \neq 0 & \text{devient } 3x - 3y \neq 0 \quad \text{puis : } x \neq y \end{array}$$

Des procédures spécifiques, qui seront précisées au paragraphe 5. 2, permettent en outre de déduire des conditions nécessaires de faisabilité. Ainsi, si ALICE connaît, parmi d'autres, les relations :

$$2 < x < 5, 0 < y < 2, 0 < z < 9$$

$$0 = 2x + y - z - 9, \text{ avec } x, y \text{ et } z \text{ entiers.}$$

ALICE est à même de déduire de nouvelles informations :

$$(x = 4 \text{ ou } x = 5) \text{ et } (z < 3)$$

Ainsi encore, dans le premier exemple sur les processeurs, en portant les valeurs du vecteur durée, dans la contrainte algébrique :

$$\forall r, r \in T \quad \sum_{\substack{t \in T \\ p(t)=r}} \text{durée}(t) \leq L$$

ALICE déduit, par sommation sur r :

$$\sum_{t \in T} \text{durées} = 68 \leq p^*.L$$

où p^* est la valeur optimale du nombre de processeurs. Avec $L = 18$, il vient : $p^* > 4$. Donnons ici, pour bien préciser cette phase, la résolution complète de l'exercice 2 sur le système booléen. Elle s'effectue, en effet, intégralement par propagation. Trouver les solutions, en variables 0-1, du système :

(1)	$0 \leq x_1x_4 + 3x_2x_5 - x_1 - 2x_3 - x_4x_5 - 1$
(2)	$0 \leq 2x_1 + 5x_3 - 2x_4 - 3x_5 - 2$
(3)	$0 \leq x_2x_4 + x_1x_4 + 2 - 5x_1$

ALICE étudie les contraintes une à une, par ordre de complexité : la contrainte (2) est jugée la plus simple, c'est-à-dire susceptible de donner le plus d'informations utiles (cf infra 5.3.2). De la contrainte (2), ALICE dérive formellement une nouvelle contrainte (Cf. infra l'algorithme 4.2) :

$$\begin{aligned} (2i) \quad 2 &\leq 2x_1 + 5x_3 \quad \text{qui est simplifié en :} \\ (2ii) \quad 1 &\leq x_1 + x_3 \quad \text{donc en : } (x_1 = 1) \text{ OU } (x_3 = 1) \end{aligned}$$

La contrainte (3) est alors analysée : elle engendre :

$$5x_1 \leq 4 \quad \text{ainsi : } x_1 = 0$$

Donc, par (2ii), il vient :

$$x_3 = 1.$$

Le système, équivalent au système de départ, est devenu :

$$\begin{aligned} (1iii) \quad 0 &\leq 3x_2x_5 - x_4x_5 - 3 \\ (2iii) \quad 0 &\leq 3 - 2x_4 - 3x_5 \\ (3iii) \quad 0 &\leq x_2x_4 + 2 \end{aligned}$$

Cette dernière contrainte est supprimée, car elle est toujours satisfaite. La première inégalité donne maintenant :

$$\begin{aligned} (1iv) \quad 3 &\leq 3x_2x_5 \text{ et aussi} \\ (1v) \quad x_4x_5 + 3 &\leq 3 \end{aligned}$$

Soit par simplification de (1iv) :

$$(x_2 = 1) \text{ ET } (x_5 = 1) \quad \text{Et par (1v) :}$$

$$(x_4 = 0) \text{ OU } (x_5 = 0) \text{ donc finalement } x_4 = 0$$

Ceci fournit, avec les résultats déjà acquis, la solution **unique** :

$$(0 \ 1 \ 1 \ 0 \ 1).$$

8.3.2 Faire des choix

Dans les programmes ordinaires, *l'énumération exhaustive* seule permet d'arriver à une solution, à un prix le plus souvent prohibitif. Dans des programmes plus évolués, de type backtrack, branch & bound ou énumération implicite (Cf. chapitre 5), l'arbre de recherche est fixé une fois pour toutes et aucune tentative n'est faite pour éviter les choix inutiles en cours de route.

ALICE, au contraire, répugne à faire des choix. Ceux-ci sont en effet coûteux : si l'espace de recherche est divisé en deux à chaque fois et si n choix doivent être faits, c'est un temps de l'ordre de 2^n qui est nécessairement atteint. Face à cette explosion exponentielle, le but d'ALICE est double : il faut, d'une part, faire des **choix qui tiennent compte du contexte**, apportent beaucoup d'information utile et tendent ainsi à minimiser l'effort de recherche pour le problème résiduel. Il faut, en outre, réduire le plus possible le nombre total de choix : ceci s'obtient, dès qu'un choix est fait, par le retour à la phase de propagation des contraintes. De plus, le plan général de la recherche contrôle continuellement l'état d'avancement de la résolution.

8.3.2.1 Différents types de choix et critères de choix

L'originalité d'ALICE est ici de **se programmer elle-même en permanence**. Le schéma de la recherche n'est pas imposé ; le système choisit lui-même à chaque pas la meilleure façon de procéder. Mais deux questions doivent dès lors être résolues :

Quels types de choix doit-on faire ?

Quels critères permettent de départager les choix possibles ?

(Seules les idées principales seront discutées dans les paragraphes suivants : pour les détails techniques, se reporter en 5.1, 5.2, 5.3.)

LES CHOIX :

Faire un **choix** correspond en fait à **l'adjonction temporaire d'une contrainte**. Intuitivement, l'espace de recherche doit alors être partagé le plus équitablement possible.

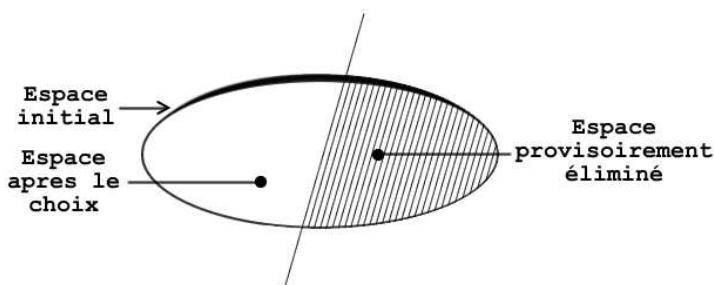


FIG. 8.2 – Effet d'un choix sur l'espace de recherche

Pour cela, il existe une première manière très classique de faire, qui consiste à fixer arbitrairement la **valeur d'une variable**. Cette méthode n'est utilisée par ALICE qu'en dernier recours, car le défaut de ce type de choix est une focalisation sur une trop petite partie de l'espace. Le système lui préfère un choix moins brutal, consistant par exemple à imposer simplement une **borne extrême** aux variations de la variable, ou encore à fixer sa **parité**. Une autre manière de procéder pour ALICE est de

raisonner à l'envers, et de considérer pour un objet, qui doit de toutes façons être atteint, l'ensemble des antécédents possibles. Les coûts éventuels interviennent ici, ALICE sachant, par exemple, se restreindre aux seuls coûts localement raisonnables. Enfin le système utilise préférentiellement toutes les contraintes disjonctives du type « de deux choses l'une, soit A, soit B » qu'il a pu mettre en évidence.

LES CRITERES :

A tout moment ALICE avance, guidée par une stratégie principale :

FAIRE CE QUI EST LE PLUS INFORMANT.

Concrètement, ceci se traduit par des critères indicateurs qui surveillent toutes les dissymétries du problème : de même qu'il est plus intéressant de traiter les contraintes à seulement une ou deux variables, quand il en existe, de même il est plus informant de faire un choix sur une variable qui ne peut prendre que peu de valeurs, ou encore qui est affectée par des coefficients élevés. Tout élément, contrainte ou variable, **qui se distingue du lot commun**, devient ainsi *un point de mire* et le système l'analyse par propagation ou par choix, suivant les procédures appropriées. Toute une batterie de critères caractérise en permanence chacune des entités intervenant dans le problème. Seul le contexte, données numériques et état de recherche, va singulariser, à l'étape courante, l'entité à traiter.

Enonçons brièvement quelques critères :

1. Nombre de contraintes où apparaît une même variable.
2. Complexité des contraintes.
3. Nombre de valeurs encore possibles pour une variable.
4. Cardinalité des ensembles dont les images doivent toutes être disjointes.
5. Valeurs des coefficients.
6. Valeur des second membres.
7. Bornes extrêmes de variation d'une variable.
8. Différence entre le coût local optimal et le coût en second.
9. Nombres d'antécédents possibles.

Tous ces critères visent à être simples et naturels. L'idée est de les considérer tous, et de ne retenir à chaque pas que le plus significatif. Mais tel critère, qui peut être excellent au début, peut aussi bien baisser en qualité et en information : ALICE sait alors l'échanger contre un autre qui va, à nouveau, permettre un arbitrage utile. Quand une heuristique devient mauvaise, elle en change. Ceci se produira à chaque fois que : une solution est atteinte, avec pour but d'en construire une meilleure ; la mesure d'évaluation sur laquelle est fondée l'heuristique perd sa signification à cause d'*ex aequo* en trop grand nombre.

Le gain effectif sur la programmation rigide classique sera mis en évidence, analysé et justifié sur un problème difficile de recherche de permutations au paragraphe 6.

8.3.2.2 Plan général de la résolution

ALICE entreprend au départ une analyse du problème dans son ensemble. Elle traduit d'abord celui-ci dans la représentation interne : les contraintes sont systématiquement

mises sous la forme la plus commode possible pour le système ; la forme « graphique » précomptée est alors préférée à la forme algébrique habituelle qui ne peut être qu’interprétée. ALICE vérifie la non-contradiction de ces contraintes et mesure la difficulté du problème. ALICE essaie en particulier plusieurs jeux de valeurs standards pour, éventuellement, trouver d’emblée une solution évidente.

Les critères potentiellement pertinents sur l’énoncé particulier sont déterminés. La propagation des informations est poussée au maximum et, si une solution n’apparaît toujours pas, la phase de choix est lancée avec retour ensuite aux propagations. Si le problème ne comporte pas d’optimisation de fonction de coût, la recherche s’arrête avec la première solution (ou avec toutes les solutions, selon la demande de l’utilisateur). **Sinon ALICE change l’ordre de ses critères de décision pour tenir compte de cette fonction.** Le système est ici plus ou moins pessimiste et vise, selon les cas, soit la faisabilité, soit l’amélioration des coûts en fonction de la difficulté mesurée du problème et du nombre de solutions estimées dans l’espace de travail. Enfin, il démontre l’optimalité à l’aide de l’un des trois arguments suivants :

1. Il a, d’une part, trouvé une borne inférieure du coût et il a su, d’autre part, exhiber une solution de valeur précisément égale à cette borne (Cf. exercice 1).
2. Il prouve que le problème initial augmenté de la contrainte (coût total inférieur à la meilleure valeur connue) n’admet aucune solution : la meilleure solution connue est optimale (Cf. exercice 1 modifié et problème sur la papeterie, paragraphe 6.2).
3. Il a entièrement étudié l’espace de recherche, la valeur optimale a donc été atteinte. Comme les choix du système sont bons, l’arbre de recherche reste souvent raisonnable et son étude implicitement exhaustive devient parfois possible (Cf. exercice 3).

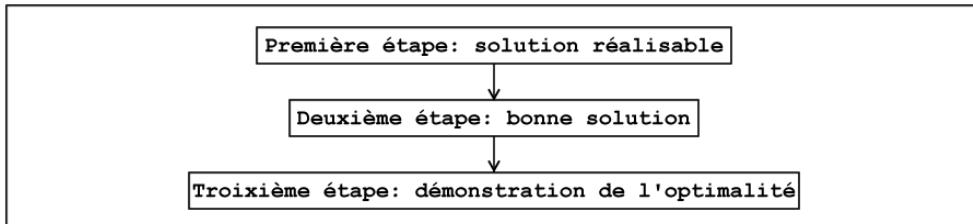


FIG. 8.3 – Plan général de résolution pour ALICE

8.3.3 Résolution des exercices

Rappel : La justification précise de la démarche d’ALICE fait l’objet du paragraphe 1.

Exercice 1

Plan : 1) Analyse de la difficulté du problème ; 2) Trouver une bonne solution ; 3) Prouver l’optimalité.

Critères pertinents : durée des tâches ; place disponible sur un processeur ; répartition équitable tâches/processseurs.

Etape 1 : Dès que le problème est mis sous forme interne, une solution évidente apparaît, une tâche est affectée par processeur. Il vient $p(1) = 1$, solution dont le coût est 9. Le problème est jugé facile et de petite taille.

Etape 2 : ALICE tente donc ici de serrer les affectations le plus possible : la tâche la plus longue va sur le processeur déjà le plus chargé, tout en respectant la limite L . Ceci donne, avec les durées :

1	2	3	4	5	6	7	8	9
6	2	8	3	6	15	6	9	13

et avec $L = 18$,

$$\begin{aligned} p(6) &= 1, & p(9) &= 2, & p(8) &= 3, \\ p(3) &= 3, & p(1) &= 4, & p(5) &= 4, \\ p(7) &= 4, & p(4) &= 1, & p(2) &= 2. \end{aligned}$$

Le coût total est donc de 4 processeurs.

Etape 3 : ALICE a calculé une borne inférieure de l'optimum, par sommation des contraintes de date limite. Le système sait que le coût optimal p^* vérifie :

$$18 \times p^* \geq \sum \text{durées} = 68 \quad \text{donc } p^* \geq 4$$

Le premier cas de preuve d'optimalité est rencontré et la résolution est terminée.

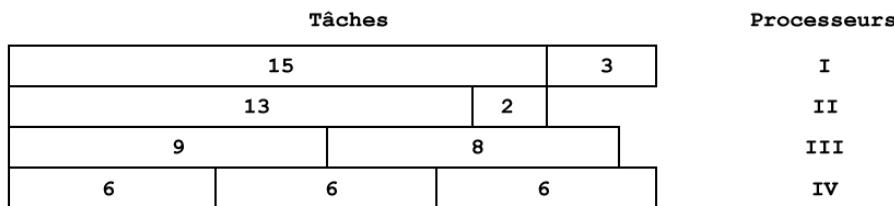


FIG. 8.4 – Optimisation de Processeurs

Nota bene :

Les données de ce petit exemple de démonstration étaient favorables. La figure ci-dessus et les connaissances d'ALICE font présager une recherche très différente si l'une des tâches de durée 6 passe à 7. En effet, si ALICE trouve encore dans ce cas la même borne inférieure de p^* égale à 4, clic ne parvient pas à placer la deuxième tâche de durée 6 sans utiliser un cinquième processeur à l'étape 2. En tout état de cause l'étape 3 impose une solution en 4 processeurs. Elle démontre qu'il n'y en a pas (troisième cas de preuve d'optimalité), en revenant sur le seul choix significatif : déplacement de la tâche de durée 8 de III en IV à nouveau les tâches de durée 6 sont implacables. L'optimum vaut bien, cette fois, 5.

Exercice 2

Il a été entièrement résolu en 3. 1. L'utilité d'introduire différents jeux de critères apparaît bien : il ne sert à rien de vouloir optimiser trop tôt quand il n'y a pas de solutions ou quand il n'y en a qu'une ! ALICE ira ainsi chercher de telles contraintes, comme plus informantes, même si elles sont noyées dans un gros problème de programmation mathématique. La **manipulation formelle des expressions** évite le recours forcé aux méthodes numériques systématiques et aveugles, couramment utilisées (simplexe, gradient).

Exercice 3

Cet énoncé est en réalité complètement « compilé » à la lecture dans la représentation graphique d'ALICE sous une forme (représentation en listes) équivalente à la suivante :

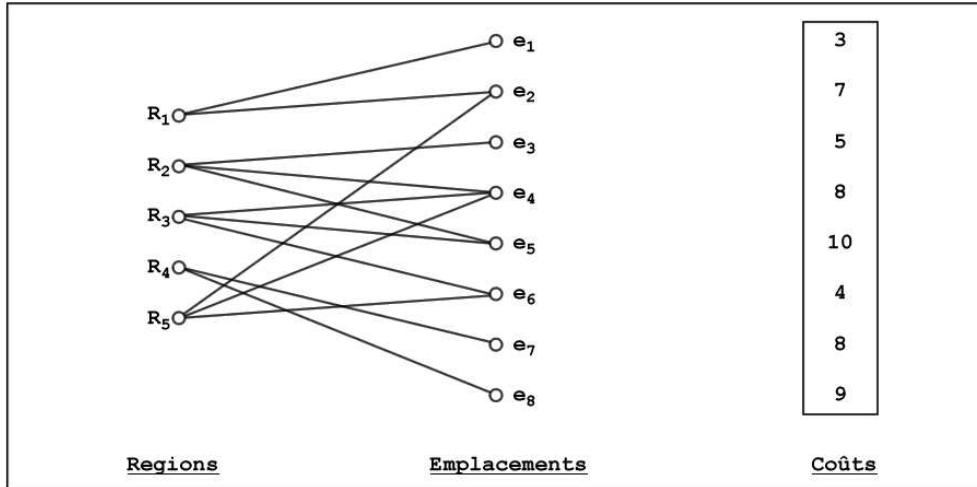


FIG. 8.5 – Localisation (représentation interne) : la région r_5 ne peut être desservie par e_2 , e_4 ou e_6

A nouveau une solution évidente est trouvée à l'étape 1 : ici ALICE retient, pour chaque région, le premier emplacement (premier successeur) possible. Soit :

$$f(1) = 1, f(2) = 3, f(3) = 4, f(4) = 7, f(5) = 2$$

Le coût total est alors $Z_0 = 3 + 5 + 8 + 6 + 7 = 29$.

A l'étape 2. les critères deviennent : coût le plus petit pour la région qui a **le moins de successeurs**. Alice pose alors :

$f(l) = 1$, $f(4) = 7$, puis $f(2) = 3$ et $f(3) = 6$, ainsi que $f(5) = 6$, avec un coût total qui tombe à $3 + 6 + 5 + 4 = 18$.

Par ailleurs, en phase 3, un coût plancher est calculé : chaque région doit être desservie : elle le sera au mieux par le plus petit coût associé à ses successeurs. ALICE considère d'abord les régions les plus contraintes (c'est-à-dire avec le moins de successeurs) : ainsi la région 4 coûte au moins 6, r_1 au moins 3, r_2 au moins 4, et r_3 coûte 4. Donc l'optimum Z^* vérifie nécessairement :

$$Z^* \geq 6 + 3 + 4 + 4, \text{ soit } Z^* \geq 17$$

$$(r_4) \quad (r_1) \quad (r_5) \quad (r_2)$$

En outre ces coûts minima peuvent à chaque fois être soustraits des coûts originaux sans changer la solution. Dorénavant ALICE travaillera donc avec les nouveaux coûts réduits :

$$(0, 0, 1, 0, 6, 0, 0, 3).$$

Etape 3

Dès lors le coût résiduel de l'emplacement 5 est prohibitif, puisque supérieur à la différence $z_0 - 17$; il en est de même pour l'emplacement 8. La solution z_0 est au plus à une unité de l'optimum. Notons que même lorsque l'optimisation n'est pas demandée, ALICE peut ainsi donner une **mesure de la distance relative à l'optimum**.

Le critère primordial devient, dans cette étape d'optimisation :

Prendre tout arc qui entraînerait le plus fort coût réduit s'il n'était pas pris.

Ici, $f(4) = 7$ est forcé. Si $f(2)$ était différent de 4, le coût total augmenterait nécessairement de 1. Or, si $f(2) = 4$, les régions 3 et 5 sont aussi couvertes. Enfin $f(1) = 1$, avec

un coût réduit nul, si bien que la construction des emplacements 1, 4 et 7 est optimale, de coût 17.

8.4 Résultats

Plus d'une centaine de problèmes très divers ont à ce jour été résolus par le système ALICE, ils concernaient d'une part des énoncés de mathématiques discrètes : couplages optimaux, équations diophantiennes, parcours Eulériens et Hamiltoniens ; d'autre part des applications réelles : distribution de courant haute tension sur le réseau EDF, organisation de plannings, tournées de livraisons, habillage d'horaires, aide à la conception en architecture comportant éventuellement une optimisation, détection optimale de pannes, affectation de disques sur des batteries dans un système d'exploitation, voyageurs de commerce et dérivés, organisation des services. Ce dernier exemple, difficile et de très grosse taille, pour lequel il n'existe aucune solution est tout particulièrement significatif de l'efficacité d'ALICE.

En outre, qui peut le plus peut le moins, une cinquantaine de puzzles et casse-têtes logiques (carrés et cubes magiques, 8 dames, puzzles de Lewis Carroll, Sam Loyd et Martin Gardner) ont été résolus. Ils montrent la grande adaptabilité du programme et la puissance du langage. La liste d'une partie des problèmes soumis se trouve en VII.7 avec plusieurs énoncés et résolutions.

On remarquera le petit nombre de choix et les faibles temps de calcul.

Après avoir expérimenté et testé en grandeur réelle ce système pendant plusieurs années, j'ai pu me convaincre de sa robustesse et de ses qualités opérationnelles. Son premier intérêt est bien, en effet d'ordre pratique. Il tient à la rapidité de la réponse entre la formulation d'un nouvel énoncé et sa résolution effective (de quelques secondes à quelques minutes). La classe de problèmes justiciables de cette approche est immense. Il ne reste qu'à faire connaître ce système auprès des utilisateurs potentiels : sociétés de service, grandes entreprises, bureaux de recherche opérationnelle, utilisateurs de logiciels en général.

Mais le second intérêt est d'ordre théorique : une voie radicalement nouvelle est ouverte à la programmation. La faisabilité d'un système où **il suffit de définir l'énoncé d'un problème** et où la résolution est entièrement « inventée » par le programme et menée à bien de manière adaptée et efficace, a été, en effet, démontrée.

8.5 ALICE : exposé détaillé

Trois points restent à préciser : la représentation interne des problèmes, le traitement des contraintes algébriques, les procédures de choix.

8.5.1 La représentation des problèmes

Habituellement un programme est conçu pour une seule application et une représentation unique suffit. Ici, il faut pouvoir représenter n'importe quel problème portant sur

des ensembles finis et des contraintes quelconques. L'idéal serait de traduire, comme dans l'exercice 3, tout l'énoncé sous forme compacte prévue à l'avance, précompilée. Cependant, pour des expressions comme, par exemple :

$$\forall i, i \in E \sum_{j \in F} a(i, j).x_j \geq b_i$$

Il n'existe pas d'autre représentation que la notation algébrique classique ci-dessus. La solution retenue pour ALICE est de représenter sous forme de « *dessins* » (graphes et hypergraphes) tout ce qui peut l'être, c'est-à-dire les contraintes simples, qui mettent au plus deux variables en jeu, et de conserver tout le reste sous la forme algébrique.

8.5.1.1 Représentation graphique

La figure 8.6 donne l'aspect général de l'organisation de l'information. Il s'agit essentiellement d'un graphe biparti : tous les points de gauche, ceux de l'ensemble D , sont les objets de départ, pour lesquels une valeur doit être trouvée. Les points de droite, ensemble I , sont les images possibles de ces objets.

SI aucune contre-indication n'est donnée dans l'énoncé, tous les arcs partant de tout u , $u \in D$, vers tout v , $v \in I$, existent potentiellement. Les informations fournies sur ces points et ces arcs - coefficients, valuations - leur sont attachées. En cours de résolution, certaines valeurs deviendront interdites. Les informations importantes sont continuellement tenues à jour : valeurs extrêmales possibles, nombres de successeurs pour tout u de D , nombre d'antécédents pour tout v de I .

Lorsque les fonctions cherchées sont d'un type particulier - injection ou surjection, option « degré imposé » minimum ou maximum des points Images - ces informations sont traduites localement : deux vecteurs associés aux points de I indiquent le nombre minimum (respectivement maximum) d'antécédents autorisés, ils prennent par exemple tous deux la valeur 1 pour une bijection.

Enfin, un hypergraphe est construit sur les sommets de l'ensemble D : il traduit, quand il y a lieu, les *disjonctions*, c'est-à-dire les contraintes qui requièrent qu'une famille de points ait, dans son ensemble, des images toutes différentes (problèmes de permutation, coloriage, planning). Ces familles (ou *hyperarêtes* : elles font intervenir *un* nombre quelconque de sommets) sont mémorisées. Une procédure spéciale est chargée, à la fin de l'interprétation de l'énoncé, de trouver l'ensemble de disjonctions le plus économique pour recouvrir D par construction de familles maximales. Cette sorte de « ramasse-miettes » (garbage collector) particulier sert à faire ensuite plus rapidement les inférences sur ces contraintes de disjonction.

8.5.1.2 Représentation formelle

L'algorithme d'unification permet de manipuler les autres contraintes. Les expressions sont en particulier normalisées grâce au règles de réécriture de la table 4. Les connectives de tête sont toujours ramenées à : \neq , \leq , ou $=$, et le second membre est toujours 0.

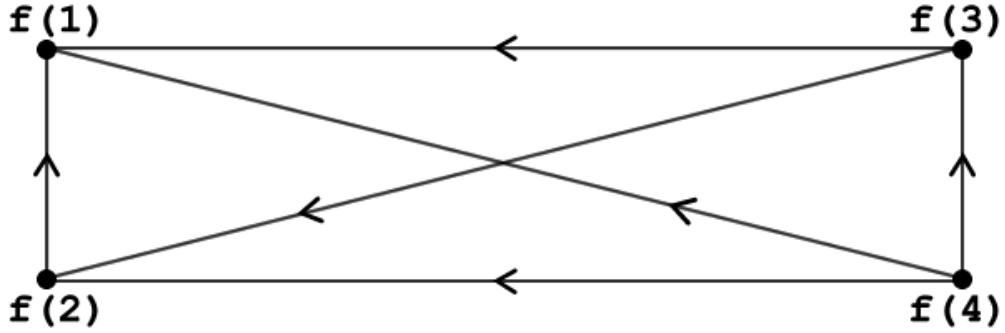


FIG. 8.6 – Schéma du graphe biparti de la représentation interne des problèmes

Dans la table 8.3, certaines unifications d'ordre deux (sur les opérateurs) peuvent être effectuées. Le symbole $0S_1$, par exemple, signifie en effet : $+$, x , $/$ ou bien modulo ; $0S_2$ signifie $=$ ou \neq ; $0S_3$ signifie $=$, \neq ou \Leftarrow ; $0S_4$ signifie $=$ ou \rightarrow (se réécrit), fff désigne une fonction quelconque.

Des filtres sont prévus, de façon à réduire le nombre d'essais d'unification. A chaque expression est associé un vecteur de caractéristiques : opérateur de tête, présence d'une constante numérique dans le premier membre, nombre d'occurrences de chaque variable. Les règles adéquates sont triées en fonction de ce vecteur. La présence d'un code CSPI indique que l'unification fait appel à une procédure de calcul numérique.

$0 * x \rightarrow 0$
$0 + x \rightarrow x$
$-(-x) \rightarrow x$
$-x > 0 \rightarrow x = 0$
$c * (x + y) \rightarrow c * x + c * y$
$-(x * y) \rightarrow -x * y$
$x * -y \rightarrow -x * y$
$x * (y * -z) \rightarrow -x * (y * z)$
$0 OS_2 c + -x \rightarrow 0 \oslash S_2 - c + x$
$0 OS_2 c + -x + y \rightarrow 0 \oslash S_2 - c + x + -y$
$0 OS_2 c_1 + (-c_2 * y) \rightarrow 0 \oslash S_2 - c_1 + c_2 * x$
$0 = c * fff x \rightarrow 0 = fff x$
true $\Rightarrow x \rightarrow x$
false $x \rightarrow$ true
true $\Leftrightarrow x \rightarrow x$
false $\Leftrightarrow x \rightarrow \neg x$
$\text{neg}(\forall x y) \rightarrow \exists x \neg y$
$\neg(\exists x y) \rightarrow \forall x \neg y$
$\neg(x \text{ OR } y) \rightarrow (\neg x) \text{ AND } (\neg y)$
$\neg(x \text{ AND } y) \rightarrow (\neg x) \text{ OR } (\neg y)$
$\neg(x = y) \rightarrow x \neg = y$
$\neg(0 \leq x) \rightarrow 0 \leq -1 + -x$
true OR $x \rightarrow$ true
false OR $x \rightarrow x$
$c \oslash S_4 c \rightarrow$ true
$c_1 \oslash S_4 c_2 \rightarrow$ false
$x > y \rightarrow 0 \leq x + -y + -1$
$x < y \rightarrow 0 \leq -x - y + -1$
$x - y \rightarrow x + -y$

$-x \oslash S 0 \rightarrow 0 \oslash S_2 x$
 $(x + y) + z \rightarrow x + (y + z)$
 $(x * y) * z \rightarrow x * (y * z)$
 $x + -x \rightarrow 0$
 $x + (-x + y) \rightarrow y$
 $x * (y/x) \rightarrow y$
 $-x * (y/x) \rightarrow -y$
 $-(x/y) \rightarrow -x/y$
 $x/-y \rightarrow -x/y$
 $x * (y/z) \rightarrow (x * y)/z$
 $x/z + y/z \rightarrow (x + y)/z$
 $x + x \rightarrow 2 * x$
 $x + (x + y) \rightarrow 2 * x + y$
 $x + c * x \rightarrow (c + 1) * x$
 $x + ((c * x) + y) \rightarrow (c + 1) * x + y$
 $-x + c * x \rightarrow (c - 1) * x$
 $-x + ((c * x) + y) \rightarrow (c - 1) * x + y$
 $c_1 * y + c_2 * x \rightarrow (c_1 + c_2) * x$
 $c_1 * x + (c_2 * x + y) \rightarrow (c_1 + c_2) * x + y$
 $-(x + y) \rightarrow -x + -y$
 $-0 \rightarrow 0$
 $1 * x \rightarrow x$
 $-1 * x \rightarrow -x$
 $(c_1 \rightarrow fff c_2) \rightarrow (fff c_2 \rightarrow c_1)$
 $(c_1 \rightarrow c_2 + fff x) \rightarrow (fff x \rightarrow c_1 + -c_2)$
 $(x \rightarrow x) \rightarrow true$
 $c_1 0S_1 c_2 \rightarrow CSP\ 1\ c_1$
 $0OS_3 c_1 \rightarrow CSP\ 2\ c_1*$
 $0OS_3 c_1 + c_2 * fff x \rightarrow CSP\ 3\ \oslash S_3\ c_1 c_2 * fff x$
 $0 = -fff x + ggg y \rightarrow CSP\ 4 = fff ggg xy$
 $coa\ x\ OS_3\ c \rightarrow CSP\ 5\ OS_3\ c\ coa\ x$
 $fff\ x\ OS_3\ c \rightarrow CSP\ 6\ \oslash S_3\ c\ fff\ x$
 $coe\ x\ \oslash S_3\ c \rightarrow CSP\ 7\ \oslash S_3\ c\ coe\ x$
 $mat\ c_1\ x\ \oslash S_3\ c_2 \rightarrow CSP\ 8\ OS_3\ c_2\ mat\ c_1\ x$
 $mat\ x\ c_1\ \oslash S_3\ c_2 \rightarrow CSP\ 9\ \oslash S_3\ c_2\ mat\ x c_1$
 $0 \Leftarrow c_1 + c_2 * x \rightarrow CSP\ 10\ c_1\ c_2 * x$
 $\sum i \in EQ \rightarrow CSP\ 11\ i \in EQ$
 $\forall i \in EQ \rightarrow CSP\ 12\ i \in EQ$
 $\exists i \in EQ \rightarrow CSP\ 13\ i \in EQ$
 $\forall i \in SUC\ j\ Q \rightarrow CSP\ 14\ i\ SUC\ j\ Q$
 $\forall i \in PRE\ j\ Q \rightarrow CSP\ 15\ i\ PRE\ j\ Q$

TAB. 8.3:

8.5.1.3 Passage de l'information entre deux représentations

- Toutes les contraintes comportant une seule variable sont éliminées de la pile des contraintes algébriques et sont mémorisées dans le graphe. Ainsi : $f(1) = 3$, $f(2) \neq 4$ ou $f(3) \geq 2$ correspondent respectivement à une affectation, à la suppression d'un arc, au déplacement d'un pointeur dans le graphe.

-
- Les relations linéaires entre variables comme $f(4) = f(5) + 6$, les inégalités du type $f(6) < f(7)$, les disjonctions comme $f(2) \neq f(4)$, sont également représentées dans ce graphe.

Ainsi, plus la résolution avance, plus les contraintes se simplifient et plus grand est le nombre d'entre elles qui sont directement traduites dans le graphe. **En fin de résolution, le graphe, seul, contient toute l'information.**

- lors d'une affectation, $f(u) = v$, tous les objets en disjonction avec u se voient interdire l'image v ; si le degré maximum de v est atteint, tous les antécédents sont supprimés;
- lors de la suppression d'un arc, vérification est faite qu'il reste des images possibles; s'il n'en reste qu'une, l'affectation correspondante est déclenchée. Il en va de même si un sommet de I atteint son degré minimum;
- lors de la traduction d'une égalité comme $x = y$, un sommet est éliminé; l'union des arêtes de disjonction et l'intersection des images sont calculées et mémorisées.

A l'occasion, une impossibilité peut être détectée lors de l'exécution récursive de ces procédures : contradictions, ensemble vide de successeurs, degré extrémal non satisfait. Celle-ci est communiquée au moniteur qui reprend le contrôle, revient en arrière et rétablit la situation du graphe et des contraintes avant le dernier choix. Notons enfin que le graphe possède les informations sur lesquelles s'appuie l'analyse locale des contraintes, qui va, maintenant, être présentée.

8.5.2 Le traitement des contraintes algébriques

Il existe un traitement général et plusieurs traitements particuliers. Nous les décrivons tour à tour et nous verrons ensuite comment ALICE choisit l'ordre de ces contraintes.

8.5.2.1 La procédure générale d'analyse d'une contrainte

Cette procédure concerne toute contrainte de la forme :

$$T_p - T_n > 0 \text{ ou bien } T_p - T_n = 0 \quad (\text{C})$$

dans laquelle tous les termes affectés d'un signe + ont été regroupés dans T_p et les autres dans T_n .

Lorsque T_p ou T_n ne comporte au plus qu'un terme, la contrainte est l'objet de l'un des traitements particuliers présentés plus loin. Dans le cas général. Il s'agit d'analyser la contrainte, en fonction des informations contenues dans le graphe, pour **engendrer de nouvelles contraintes nécessairement satisfaites** pour que la contrainte originale le soit aussi.

Ces contraintes seront, par construction, plus simples, et susceptibles de fournir une information directement utilisable. Les intervalles de variations des deux sommes T_p et T_n résument l'information principale. Pour que la contrainte (C) soit satisfait, il faut que ces deux intervalles présentent une disposition relative en accord avec la connective principale de (C).

Par standardisation, ALICE se ramène toujours à des variables positives. La procédure d'analyse s'écrit alors facilement en donnant des noms à ces intervalles, ainsi qu'à leur intersection :

Soient $[P, PP]$ pour T_p et $[N, NN]$ pour T_n ces intervalles, avec :
 $A = \sup(P, N)$ et $B = \inf(PP, NN)$ soit : $[A, B] = [P, PP] \cap [N, NN]$.

Il vient :

```

SI A > B
ALORS
    DEBUT (L'intersection est vide)
        SI A = P ET SI l'opérateur est ≥
            ALORS (C) est trivialement satisfait. Tuer (C).
            SINON (C) ne peut être satisfait. ECHEC. FIN

SI A = B
ALORS
    DEBUT (L'intersection est réduite à un point)
        SI A = P ET SI l'opérateur est ≥
            ALORS (C) est vraie. Tuer (C).
            SINON ajouter deux nouvelles contraintes :  $T_p = A$  et  $T_n = A$ 
        FIN

SI A < B ALORS      DEBUT (Cas général)
    SI P < N ALORS ajouter :  $T_p \geq A$ 
    SINON SI l'opérateur est = ALORS ajouter :  $T_n \geq A$ 
    SI PP < NN ALORS ajouter :  $T_n \leq B$ 
    SINON SI l'opérateur est = ALORS ajouter :  $T_p \leq B$ 
FIN

```

Algortihme 4.2

8.5.2.2 Cas particuliers

1) Le premier est celui où T_p , ou bien T_n est vide ; alors (C) est trivialement vraie ou fausse. Si l'une de ces sommes ne comprend qu'un terme, la procédure est affinée et engendre autant de contraintes qu'il existe de termes dans l'autre somme. Si T_n est réduit à un terme, ALICE écrit, pour tout t de T_p la nouvelle contrainte :

$$t \geq NN - \sum_{\substack{t' \in T_p \\ t' \neq t}} \max t'$$

Exemple :

La contrainte $2x_1 + 2 - 9 > 0$ donne ainsi, si $x_1 \in [2, 5]$ et $x_2 \in [0, 2]$

$$2x_1 \geq 9 - 2 \text{ et } x_2 \geq 9 - 10$$

soit $x_1 \geq 4$ et rien de nouveau pour x_2 .

2) Le cas des équations diophantiennes de la forme : $ax + by = c$ est traité à part, à l'aide d'une procédure de congruence classique. Les valeurs marquées interdites dans le graphe sont ici particulièrement utiles. Les facteurs entiers communs sont recherchés

dans toutes les contraintes en égalité et des équations de congruences sont, là aussi, engendrées.

Exemple :

La contrainte $15x + 18y - 4z - 6 = 0$ donne ainsi naissance à :

$$x \equiv 0 \pmod{2} \text{ et } z \equiv 0 \pmod{3}$$

qui sont immédiatement traduites dans le graphe par suppression des arcs qui ne satisfont pas ces congruences.

3) Les égalités permettent des éliminations de variables. Cependant, ces éliminations ne sont entreprises que dans le cas linéaire et lorsque chaque variable apparaît au moins deux fois dans l'ensemble des contraintes (dans le cas contraire, de telles substitutions ne sauraient fournir d'information supplémentaire).

Exemple :

$$\left\{ \begin{array}{l} x + y = 2 \\ x + 2y + 3z \geq 3 \end{array} \right. \quad \begin{array}{l} x \rightarrow 2 - y \\ y + 3z \geq 1 \end{array}$$

donne naissance à :
et donc à :

4) Il reste enfin un type de contraintes dont nous n'avons pas parié : ce sont toutes les contraintes implicites où le nom exact des objets n'est pas connu. Par exemple :

$$\forall v, v \in I \quad \sum_{\substack{u \in D \\ f(u)=v}} c(u) \leq b \quad (1)$$

Nous avons déjà vu que ces contraintes pouvaient être cumulées pour donner naissance à des conditions globales nécessaires. ALICE les utilise en outre comme tests d'élimination. D'une façon générale, toute contrainte de ce type, qui exprime une condition sur un ensemble W d'objets w , vérifiant une propriété $Q(w)$ est analysée à chaque modification des affectations dans W . Une telle contrainte s'écrit :

$$\sum_{w:Q(w)} c(w) \leq b$$

où l'écriture $w : Q(w)$ signifie l'ensemble des w qui satisfont la propriété Q . Si, à un instant donné, il existe w_0 pour lequel on sait que :

$$c(w_0) + \sum_{\substack{w:Q(w) \\ w \neq w_0}} > b$$

alors, w_0 ne saurait satisfaire la propriété Q .

Exemple :

Ainsi, dans le cas précédent, et lors de toute affectation, $f(u_0) = v_0$, l'inégalité (1) est étudiée pour v_0 : le coût k des objets qui ont actuellement v_0 pour image est calculé :

$$k = \sum_{f(u)=v_0} c(u) \quad (8.1)$$

Pour tout u_1 tel que $c(u_1) > b - k$, la propriété $f(u_1) = v_0$ ne peut être vérifiée. Les arcs correspondants sont éliminés du graphe.

8.5.2.3 Ordre de traitement des contraintes

L'ensemble des contraintes dans ALICE est dynamique, et ceci à double titre. En premier lieu, comme on l'a vu, et contrairement aux programmes classiques en Recherche Opérationnelle ou en algorithmique numérique, des contraintes sont, en permanence ajoutées ou retranchées, d'autres poussent d'une représentation à une autre. En second lieu, le système traite, à un instant donné, la contrainte qu'il estime la plus intéressante, suivant une procédure qui va maintenant être décrite.

Tout d'abord, priorité est donnée au graphe ; un **agenda de tâches** est créé, et, tant qu'il n'est pas épuisé, les procédures d'inférence sur le graphe sont seules au travail. Lorsque le contrôle est donné au traitement algébrique, les contraintes qui ont reçu le plus d'information depuis leur dernière analyse, sont recherchées. En particulier, toutes celles qui ont vu l'une, au moins, de leurs inconnues recevoir une valeur, sont prioritaires. S'il n'y en a pas, c'est le nombre de valeurs éliminées par le graphe qui joue.

Pour départager ces contraintes ALICE mesure leur difficulté : plus une contrainte est difficile à satisfaire, plus elle est contraignante et plus elle peut apporter d'information.

Pour toute contrainte algébrique (C) de la forme $T_p - T_n \geq 0$, la *difficulté* $d(C)$ s'écrit :

$$d(C) = \frac{\max(0, b)}{\max(1, S - b)}$$

formule empirique dans laquelle b désigne le terme constant de T_n et S la somme des coefficients des termes positifs divisés par les degrés des termes. Plus b est grand, et plus S est petite, plus il sera facile de trouver un jeu de valeurs pour les inconnues satisfaisant la contrainte (C). ALICE exploite, de plus, le fait que toute variable de T_n est susceptible de fournir un renseignement sur une variable de T_p . Plus précisément la borne supérieure d'une variable de T_n constraint les bornes supérieures des variables de T_p , et réciproquement.

L'analogue d'un petit dessin, représentatif en réalité d'une *fonction de Grundy*, peut donc être construit, qui traduit l'ordre logique dans lesquelles les informations sur ces bornes vont circuler de manière agréable.

Les bornes inférieures, puis supérieures, sont considérées. Les contraintes portant sur des variables qui n'attendent aucun autre renseignement sont d'abord étudiées. Puis, les nouvelles bornes calculées se propagent par l'étude de celles qui ne dépendent que de celles déjà connues. Illustrons ceci par le cas de contraintes linéaires. Supposons qu'ALICE ait décidé de considérer, parmi les autres, les trois contraintes :

$$\begin{aligned} (C_1) \quad & f(l) + f(2) - f(3) + f(4) - 3 \geq 0 \\ (C_2) \quad & f(1) - f(2) - 1 \geq 0 \\ (C_3) \quad & f(3) - f(4) - 2 \geq 0 \end{aligned}$$

Le diagramme ?? montre les dépendances, marquées par des flèches entre les bornes inférieures et supérieures pour les différentes inconnues il existe un arc de $f(i)$ vers $f(j)$ appartenant au terme T_p dans une même contrainte (C).

Par exemple, ALICE sait que la borne inférieure de $f(3)$ donne des renseignements sur les bornes inférieures de $f(2)$ et $f(1)$. Dans ce dessin, un sommet - $f(4)$ - n'attend rien de $f(3)$, de $f(2)$ et de $f(l)$: il n'a pas d'antécédent : ALICE traite d'abord la contrainte la plus simple où $f(4)$ est, en partie T_n , soit la contrainte (C_3).

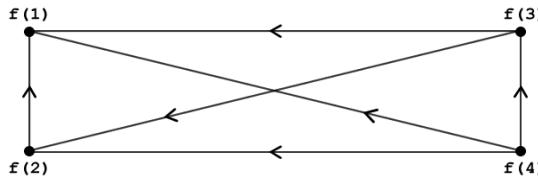


FIG. 8.7 – Fonction de Grundy

Supposons que la borne inférieure actuelle de $f(4)$ soit 5. La contrainte (C_3) implique $f(3) \geq 7$. Les termes T_n de (C_1) n'ont alors plus d'antécédents non étudiés ; c'est cette contrainte (C_1) qui est analysée. Elle donne :

$$f(l) + f(2) \geq 3 + 5 + 7$$

Supposons que le graphe fournisse en outre : $f(1) \leq 8$
 Alors, il vient : $f(2) \geq 7$

Finalement toutes les contraintes ont été prises dans le bon ordre, de manière à fournir l'information maximale : la dernière, (C_2), conduite : $f(1) = 8$ avec finalement $f(2) = 7$, $f(3) = 7$, $f(4) = 5$. Un graphe et une étude similaire traitent le cas des bornes supérieures et donne de nouvelles informations. ALICE reste le plus longtemps possible dans la phase d'inférence, passant alternativement du graphe aux procédures algébriques, **retardant les choix le plus longtemps possible**. Cependant, lorsqu'aucune nouvelle information ne peut être déduite, une hypothèse doit être faite, contraignant un peu plus le problème et permettant d'avancer dans sa résolution : ALICE procède alors nécessairement à un choix.

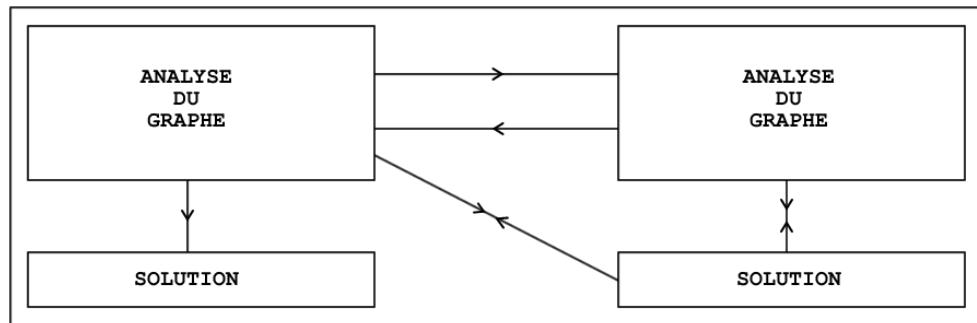


FIG. 8.8 – Schéma global de contrôle pour ALICE

8.5.3 Les procédures de choix

Sept types différents de choix sont effectivement disponibles. Ils sont donnés ci-après dans **l'ordre de préférence du système**.

- 1 : Affirmer l'une des branches de l'alternative d'une contrainte OU,
- 2 : Fixer la valeur de la borne inférieure ou supérieure d'une variable,
- 3 : Imposer une limite au coût d'une affectation,
- 4 : Enumération locale exhaustive sur une équation à 2 ou 3 variables,
- 5 : Enumération globale, lorsque la taille du problème passe au dessous d'un seuil fixé,
- 6 : Forcer l'antécédent d'un point de l'ensemble image,
- 7 : Choisir l'image d'un point de l'ensemble de départ.

8.5.3.1 Choix du type de choix

Type 1 : Affirmer l'une des branches de l'alternative d'une contrainte OU. Chaque fois qu'une des branches d'une contrainte de type OU comporte au plus deux variables. ALICE choisit de la supposer vraie.

Type 2 : Fixer la valeur de la borne inférieure ou supérieure d'une variable. Un choix de ce type est effectué sur la variable qui intervient dans le plus grand nombre de contraintes, et dont l'ensemble image est le plus grand. Cet ensemble est alors coupé en deux. Ce choix n'a de sens que lorsque l'ensemble image est un ensemble de nombres.

Type 3 : Imposer une limite au coût d'une affectation. Les choix de type 3 sont optimistes : ils interviennent à l'étape 2 lorsqu'une solution évidente a déjà été trouvée. Les coûts les plus forts sont tour à tour éliminés, jusqu'à ce que pour chaque point de D , l'ensemble des arcs ait ainsi été réduit de moitié.

Types 4 et 5 : Enumération locale exhaustive sur une équation à 2 ou 3 variables. Enumération globale, lorsque la taille du problème passe au-dessous d'un seuil fixé. Les choix 4 et 5 font appel à la puissance de calcul même de l'ordinateur : il est inutile de tenter des manipulations formelles sophistiquées quand le problème est devenu de petite taille. Moins il y a de contraintes et moins il y a d'arcs de D vers I plus ALICE a tendance à lancer ces procédures d'énumération. Notons que pour évaluer cette taille limite. **ALICE sait ce qu'elle sait faire** et en tient compte : les contraintes d'égalité sont liées au plus grand nombre de moyens formels d'analyse, alors qu'au contraire, les contraintes d'inégalité, et a fortiori les contraintes OU ne relevant pas du type 1, sont moins manipulables. Plus ces deux derniers types de contraintes sont nombreux, plus vite est atteinte, pour ALICE, la taille limite qui déclenche l'énumération.

Types 6 et 7 : Forcer l'antécédent d'un point de l'ensemble image. Choisir l'image d'un point de l'ensemble de départ. Il s'agit ici de déterminer un couple (Objet de D , image de I). Ces procédures ne sont appliquées qu'en dernier ressort, donc lorsque l'espace de recherche a déjà été considérablement réduit.

Préférence est donnée au choix 6, toutes les fois que : l'ensemble 1, tout entier, doit être atteint ; il existe un point de I qui a moins d'antécédents que le plus petit nombre de successeurs des points de O : ALICE donne la priorité au point le plus contraint. Dans les deux cas, ALICE utilise toute une batterie de critères pour choisir le couple (point de départ, point image) qui apportera le plus d'informations dans la suite de la résolution du problème.

Un critère particulier ne sera retenu à un instant donné de l'exécution que s'il est discriminant pour son ensemble de référence. ALICE décide qu'un critère est discriminant

si, et seulement si, m étant la note du meilleur candidat pour ce critère, moins d'un quart des candidats ont une note supérieure à $m - \frac{m}{4}$. Cette procédure élimine ainsi les critères plats (toutes les notes sont voisines) ou les critères lâches (trop d'ex-aequo avec des notes élevées). Le sous-ensemble des meilleurs candidats désignés par un critère est ensuite filtré par un second critère discriminant et ainsi de suite. Jusqu'à ce qu'il ne reste plus qu'un candidat.

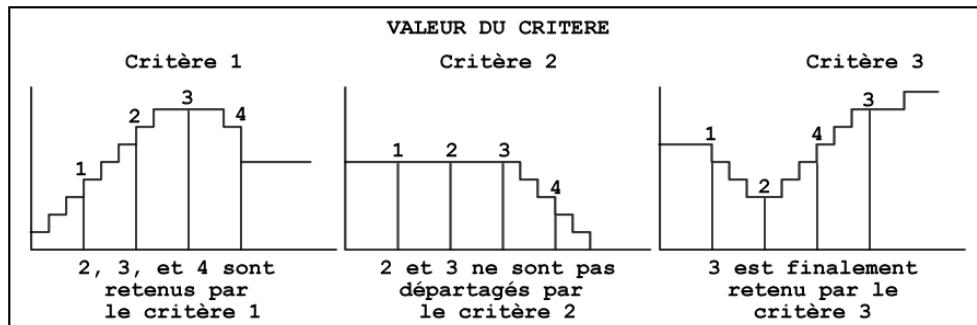


FIG. 8.9 – Valeurs de trois critères pour 4 variables. Choix d'ALICE : variable 3

Nous donnons maintenant la liste des critères utilisés par ALICE. Elle joue à la fois sur les représentations graphiques et algébriques. Nous donnerons ensuite les règles de production qui déterminent la stratégie, c'est à dire l'ordre dans lequel, suivant le contexte, ces critères sont examinés par ALICE.

Liste des critères pour le choix d'un point

- a) Minimum de successeurs ;
- b) Maximum de points en disjonction ;
- c) Maximum de contraintes algébriques où ce point apparaît ;
- d) Minimum de la difficulté globale sur ces contraintes ;
- e) Maximum des coefficients sur ce point ;
- f) Maximum du minimum des valueations des arcs successeurs* ;
- g) Maximum du *regret** ;
- h) Maximum de *l'intérêt** ;

Liste des critères pour Le choix d'un point de l'ensemble image

- α) Nombre d'antécédents potentiels* ;
- β) Nombre d'antécédent réels* ;
- γ) Valeur numérique de ce point ;
- δ) Valeur des coefficients de ce point ;

(*) Voir les explications ci-dessous. La *difficulté* d'une contrainte a été définie en 5.2.3. La difficulté globale des contraintes sur une variable de D est simplement la somme des difficultés élémentaires. Tous les critères ont pour but de placer en tête les points les plus difficiles. Ainsi (f) s'intéresse aux points dont le coût minimum possible est le plus grand. Le critère (g) intervient dans les problèmes d'optimisation.

Le regret d'un point est la différence entre le coût le plus bas et le coût qui le suit immédiatement : **plus ce regret est élevé, plus le point est critique.**

Le critère (h) concerne les contraintes algébriques. *L'intérêt* d'une variable x est défini par la somme, sur l'ensemble des contraintes où figure x , des variations des difficultés

lorsque x passe de sa valeur minimale à sa valeur maximale. L'intérêt de x croît lorsque la difficulté globale des contraintes sur x diminue.

Le critère (α) concerne les antécédents u , $u \in D$, encore possibles pour tout point v de I , tandis que les antécédents réels du critère (β) sont les points u de D qui ont déjà v pour image.

8.5.3.2 Ordonnancement des critères et stratégie d'ALICE

L'ordre dans lequel les critères sont classés est choisi par ALICE elle-même, il dépend d'une part de la nature du problème (dès l'énoncé, ALICE sait que certains critères ne peuvent être pertinents) et, d'autre part, de l'état d'avancement de la solution.

Deux méta-heuristiques tout à fait générales ont permis d'écrire les règles stratégiques qui suivent, il s'agit de :

- H1) faire les choix les plus informants.
- H2) faire les choix les moins coûteux.

Les règles de production pour fixer l'ordre des critères qui en découlent sont les suivantes :

- SI étape = 0 (Lecture de l'énoncé)
 - ET les inconnues sont booléennes
 - | ALORS supprimer le critère (a).
 - FSI
- SI aucune disjonction n'est imposée
 - OU SI tous les sommets ont le même nombre de disjonctions
 - | ALORS supprimer le critère (b).
 - FSI
- SI il n'existe aucune contrainte algébrique
 - | ALORS supprimer les critères (c), (d) et (h).
 - FSI
- SI les contraintes sont toutes algébriques
 - ET les contraintes sont linéaires
 - | ALORS prendre les formules simplifiées pour calculer les difficultés, faire passer en tête le critère (h).
 - FSI
- SI le problème n'est pas un problème d'optimisation
 - OU SI il n'y a pas de valuation sur les arcs $D \times I$
 - | ALORS supprimer les critères (f) et (g).
 - FSI
- SI il n'existe pas de coefficient sur D
 - | ALORS supprimer le critère (e).
 - FSI
- SI il n'existe pas de coefficient sur I
 - | ALORS supprimer le critère (δ).
 - FSI
- SI tous les points de I doivent être atteints ($DMI \geq 1$)
 - | ALORS positionner le critère (α) à minimum.
 - FSI
- SI des degrés maximum sur I sont imposés ($DMA = 0$)

	ALORS prendre comme formule pour (a) : nombre d'antécédents (v) – degré maximum (v) positionner le critère (α) à maximum
FSI	
• SI	le problème est une optimisation
ET	la minimisation d'une fonction de $f(D)$ intervient
	ALORS positionner le critère (β) à maximum
	SINON positionner le critère (β) à minimum
FSI	
• SI	il existe des contraintes en < (respectivement >)
ET	ces contraintes portent sur la valeur numérique (resp. sur les coefficients) des points de D
	ALORS positionner le critère (γ) (resp. (δ)) à maximum (resp. minimum)
	SINON supprimer le critère (γ) (resp. (δ))
FSI	
• SI	étape = 1 (recherche de la faisabilité du problème)
ET	S'il n'existe pas de solution triviale
	ALORS placer en tête les critères (a), (b), (c), (d) et (e)
FSI	
• SI	étape = 2 (construction d'une bonne solution)
	ALORS faire les choix de type 3, placer en tête les critères (f), (g) et (h).
FSI	
• SI	étape = 3
	ALORS
	SI le nombre total de retours arrière est inférieur à D
	ALORS placer en tête les critères (f), (g) et (h).
	SI le nombre de retours arrière est inférieur à $2 \times D$
	ALORS placer en tête les critères (a), (b), (g) et (h), SINON ne conserver que les critères (a), (b), (c), (d) et (e).
FSI	
FSI	

Ainsi, dans la suite des étapes, le programme est d'abord « méfiant », puis « optimiste » dès qu'une solution a été trouvée, et à nouveau « méfiant » lorsque le nombre de choix ayant échoué augmente, et que le problème est de plus en plus contraint. Soulignons que, d'un côté ces critères et leur ordonnancement ne font qu'exprimer des **heuristiques**, mais que d'autre part, par construction. ALICE fournit des **solutions parfaitement optimales**. Autrement dit, il ne saurait y avoir de contradiction entre heuristique et rigueur. La preuve de l'optimalité des solutions obtenues par ces heuristiques est donnée par ALICE suivant les moyens que nous allons exposer.

8.5.3.3 Preuves d'optimalité

Dès qu'une solution, de coût total Z_0 , est obtenue, la contrainte $Z < Z_0$ est ajoutée à tout problème de minimisation dont Z est la fonction économique. Si celle-ci est un polynôme, cette contrainte vient simplement s'ajouter aux contraintes algébriques de départ. La preuve de l'optimalité est faite lorsqu'un problème impossible est rencontré. Deux cas particuliers fréquents sont traités différemment. Ils concernent les fonctions

économiques :

$$Z_1 = \min_f \sum_{u \in D} \text{coût } (u, f(u))$$

rencontrées à chaque fois que les arcs d'affectations sont valués (en particulier dans les problèmes de voyageurs de commerce, de tournées, de partitions), et :

$$Z_2 = \min_f \max_{u \in D} a(u) * f(u)$$

problèmes dits « bottleneck » où seul le plus fort coût, pondéré par un coefficient $a(u)$, intervient (problèmes d'ordonnancement, de coloration, de découpe).

Cas Z_1 . ALICE procède à l'étape 3 à la réduction habituelle des coûts. Chaque point u de D entraîne un coût minimal égal à :

$$\min_v \text{coût } (u, v)$$

Cette quantité est retranchée à tous les coûts des arcs issus de u . de telle sorte que le coût minimal pour chaque point soit nul. Si, par la suite, l'arc correspondant vient à être supprimé, ALICE effectuera une nouvelle réduction des coûts. La somme totale ainsi ôtée constitue une borne inférieure \underline{Z} de la solution en cours. L'optimalité est prouvée par élimination des arcs de coût supérieur à $Z_0 - \underline{Z}$ jusqu'à ce qu'un point n'ait plus de successeur.

Cas Z_2 : Pour obtenir une solution meilleure, au sens strict, qu'une solution connue de valeur Z_0 , les images v doivent vérifier pour tout u de D :

$$v < a(u) * Z_0$$

Cette inégalité réduit d'autant l'ensemble des images possibles en chaque point. En outre, une borne inférieure \underline{Z} de l'optimum est calculée à partir de toute contrainte du type :

$$\forall v, v \in D, P(u, v) \Rightarrow \sum Q(u) \leq k \quad (8.2)$$

comme dans l'exemple 1.

Enfin, si les degrés maximum, DMA(v), sont imposés pour les points images, ALICE en déduit également la borne inférieure \underline{Z} , où \underline{Z} est le premier entier w tel que la somme :

$$\sum_{v=1}^w \text{DMA}(v)$$

soit supérieure à la cardinalité de D : tout point de D doit recevoir une image.

Tous ces calculs de bornes sont cruciaux dans la mesure où ils permettent, suffisamment tôt, l'arrêt de la recherche et la preuve de l'optimalité. Remarquons qu'ils sont toujours bâtis sur une condition nécessaire que doit vérifier toute solution. Ils sont donc effectués en plongeant l'espace de recherche dans un espace plus grand, moins contraint (cf. figure 8.10).

Ainsi, par construction, toute solution de R est au moins aussi bonne que la meilleure solution de E .

Cette démarche de calcul de borne inférieure est donc duale du processus de choix. A eux deux. Ils fournissent un encadrement du minimum réel Z^* :

$$\underline{Z} \geq Z^* \geq Z_0$$

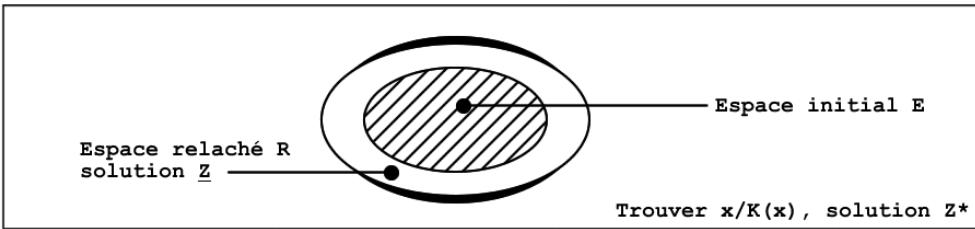


FIG. 8.10 – Restriction de l'espace de recherche

8.6 Exemples détaillés de résolutions

8.6.1 Un casse-tête logico-arithmétique

« *François et Georges jouant au tennis.
François bat Georges six jeux à trois.
Quatre fois, celui qui était au service a perdu le jeu.
Qui a servi le premier ?*

Nous avons besoin, pour poser le problème, d'un vecteur indiquant qui a servi le jeu j ($j = 1$ à 9) et d'un autre indiquant qui l'a gagné. Posons :

$$\begin{cases} \text{FS}(j) = 1 \text{ si François sert le } j\text{-ième jeu, 0 sinon;} \\ \text{GS}(j) = 1 \text{ si Georges sert le } j\text{-ième jeu, 0 sinon;} \\ \text{FG}(j) = 1 \text{ si François gagne le } j\text{-ième jeu, 0 sinon;} \\ \text{GG}(j) = 1 \text{ si Georges gagne le } j\text{-ième jeu, 0 sinon.} \end{cases}$$

Les règles du tennis imposent :

$$\begin{cases} \text{FS}(j) = 1 - \text{GS}(j) & (\text{F ou G sert}) \\ \text{GS}(j) = 1 - \text{GS}(j-1) \text{ pour } j \neq 1 & (\text{F et G servent à tour de rôle}) \\ \text{FG}(j) = 1 - \text{GG}(j) & (\text{F ou G gagnent un jeu}) \\ \text{FG}(9) = 1 & (\text{F a gagné le dernier jeu}) \end{cases}$$

L'énoncé complet en ALICE est donc :

SOIT ENSEMBLE JEUX= $\llbracket 1, 9 \rrbracket$
 BOOL= $\llbracket 0, 1 \rrbracket$

TROUVER FONCTIONS FS, GS, FG, GG JEUX \longrightarrow BOOL

$$\begin{aligned} \text{AVEC } \forall j \quad j \in \text{JEUX} \quad (\text{FS}(j) = 1 - \text{GS}(j)) \wedge & \quad (1) \\ ((j \neq 1) \implies \text{GS}(j) = 1 - \text{GS}(j-1)) \wedge & \quad (2) \\ (\text{FG}(j) = 1 - \text{GG}(j)). & \quad (1) \end{aligned}$$

$$\text{AVEC } \sum_{j \in \text{JEUX}} \text{FG}(j) = 6 \quad (4)$$

$$\text{AVEC } \sum_{j \in \text{JEUX}} [\text{FS}(j) * \text{GG}(j) + \text{GS}(j) * \text{FG}(j)] = 4 \quad (5)$$

(contraintes sur les services perdus)

$$\text{AVEC } \text{FG}(9) = 1 \quad (6)$$

FIN

Le problème peut être résolu **sans aucune énumération**, comme le fait ALICE. Les variables $FS(j)$ et $FG(j)$ sont tout d'abord éliminées par (1) et (3) ; leurs expressions sont reportées dans (4) et (5), qui deviennent :

$$\sum GG(j) = 3 \quad (4i)$$

$$\sum (GS(j) - 2 \cdot GS(j) \cdot GG(j) + GG(j)) = 4 \quad (5i)$$

Par ailleurs, tous les $GS(j)$ sont obtenus à partir du seul $GS(1)$, à l'aide de la relation (2) :

$$GS(j) = 1 - GS(1) \text{ pour tout } j \text{ pair}$$

$$GS(j) = GS(1) \text{ pour tout } j \text{ impair}$$

Portant ces relations dans l'égalité (5i), ALICE obtient d'abord :

$$5 \cdot GS(1) + 4 - 4 \cdot GS(1) - 2 \cdot \sum (GS(j) \cdot GG(j)) + \sum GG(j) = 4 \quad (5ii)$$

Puis, en utilisant la relation (4i) :

$$GS(1) - 2 \cdot \sum (GS(j) \cdot GG(j)) + 3 = 0 \quad (5iii)$$

L'analyse par le module congruence de cette égalité impose alors : $GS(1)$ impair, ce que le graphe traduit en : $GS(1) = 1$. C'est donc Georges qui a servi le premier.

REMARQUE : La somme $\sum (GS(j) \cdot GG(j))$ vaut donc 2 et, puisque $GS(j) = 1$ si j est impair, le nombre de jeux impairs gagnés par Georges vaut 2 également : comme le neuvième jeu a nécessairement été gagné par François, il y a, pour Georges $4 * 3/2$ possibilités de gain sur les jeux impairs et 4 sur les jeux pairs, soit au total 24 parties possibles pour la situation annoncée, bien qu'ALICE donne la réponse sans aucun choix.

8.6.2 Un problème de fabrication de papier

« Une entreprise de fabrication de papier doit honorer des commandes relatives à n qualités différentes de papiers. Chacune d'entre elles peut être obtenue sur l'une quelconque des M machines à papier de l'entreprise. Cependant, comme les coûts de changement de pâte et de réglage sont grands devant ceux de la seule fabrication, chaque qualité doit impérativement être fabriquée **entièvement** sur la même machine. Les quantités demandées en tonnes, $q(i)$ sont connues pour chaque qualité ($i = 1$ à n) de même que la production totale maximale $p(j)$ de chaque machine ($j = 1$ à m). Enfin le coût total, $c(i, j)$, de fabrication du papier de qualité i sur la machine j a été évalué pour tout couple (i, j) . On demande d'organiser sous ces contraintes la production complète au moindre coût. »

Application numérique (cas réel proposé par H.Sndl), $n = 10$ qualités, $m = 4$ machines. L'énoncé de ce problème en langage ALICE est immédiatement :

$p(j)$	1	2	3	4	5	6	7	8	9	10	
	↓	3	5	10	1	2	3	5	3	8	2
1	9	13	9	1	3	14	15	7	9	15	8
2	10	13	9	20	6	17	7	9	16	17	1
3	20	18	15	11	14	13	8	8	12	17	12
4	3	5	4	19	2	6	15	19	4	15	20

\downarrow
 $c(i, j)$

L'énoncé de ce problème en langage ALICE coule de source :

SOIT	l'ensemble Papiers = $\llbracket 1, n \rrbracket$
	Machines = $\llbracket 1, m \rrbracket$
le coefficient	q sur Papiers
	p sur Machines
	c sur Papiers * Machines

TROUVER la fonction $f : \text{Papiers} \rightarrow \text{Machines}$

AVEC $\forall j \in \text{Machines} :$

$$\sum_{\substack{i \in \text{Papiers} \\ f(i)=j}} q(i) \leq p(j)$$

AVEC $\min_f \sum_{i \in \text{Papiers}} c(i, f(i))$

FIN

10,	4								
4,	5,	10,	1,	2,	3,	4,	3,	8,	2.
9,	10,	20,	3.						
13,	9,	1,	3,	14,	15,	7,	9,	15,	8.
13,	9,	20,	6,	17,	7,	9,	16,	17,	1.
18,	15,	11,	14,	13,	8,	8,	12,	17,	12.
5,	4,	19,	2,	6,	15,	18,	4,	15,	20.

Pour le programme, le premier travail dans la première phase est de chercher s'il existe une solution réalisable, c'est-à-dire une fonction de fabrication f qui satisfait toutes les contraintes sauf celle d'optimisation :

$$\left(\min_f \sum_{i \in \text{Papiers}} c(i, f(i)) \right)$$

La contrainte de capacité en $\sum q(i) \leq p(j)$, qui commence par un quelque soit, fait naître une condition nécessaire de faisabilité qui s'obtient simplement en cumulant cette contrainte sur l'ensemble qui est quantifié, soit ici l'ensemble des machines. Il vient alors :

$$\sum_{i \in \text{Papiers}} q(i) \leq \sum_{j \in \text{Machines}} p(j)$$

Constatant qu'avec les données numériques fournies, les deux sommes sont égales -elles valent toutes deux 42 - le programme en déduit :

- Qu'il n'y a pas d'impossibilité a priori à la faisabilité.
- Que le signe $<$ doit, pour tout j , être changé en une égalité rigoureuse : aucune chute n'est nulle part autorisée.
- Que certaines affectations sont néanmoins interdites, la valeur d'un $q(i)$ dépassant celle du $p(j)$ associé. Dans ce cas on a : $q(i) > p(j) \Rightarrow f(i) \neq j$.

Il en va ainsi de $f(3) \neq 1$ et de $f(2), f(3), f(7), f(9) \neq 4$. Aucune implication ne peut plus être obtenue et le programme doit faire un choix. Un choix sur une contrainte OU ne peut être effectué car aucune contrainte de ce type n'est présente ici. Les choix sur les petites valuations seraient possibles, mais sont éliminés par le programme qui juge l'espace de recherche suffisamment petit ($4^{10} = 1.048.576$) pour faire un choix plus précis de type $f(i) = j$. Rappelons ici les critères dont le programme dispose pour définir un tel choix.

Critères pour le choix du sommet antécédent i

- | | |
|-------------------------------------|--------------------------|
| a : min successeurs | e : max max coefficients |
| b : max disjonctions | f : max min valuation |
| c : max nombre contraintes avec i | g : max regret |
| d : max difficultés contraintes | h : intérêt |

Le programme va déterminer à chaque choix le papier à considérer en fonction de ces critères. Mais, d'emblée. Il élimine trois critères - b, c et d - qui sont sans objet pour ce problème : les valeurs associées sont toutes nulles pour tous les i . Ensuite les critères d'optimisation - f, g et h - ne sont pas pris en compte dans la première phase. Deux critères - a et e - subsistent. La distribution des valeurs pour le critère e (10 8 5 5 ...) est moins significative que celle du critère a : en effet seule la qualité 3 possède 2 successeurs et la distribution vis-à-vis de ce critère est (23334...). La qualité de papier 3 vient finalement en tête car le critère e la départage alors des qualités 2, 7 et 9. Le choix de la machine j se fait a priori avec le jeu de critères :

Critères pour le sommet successeur j :

- | |
|--|
| α : nombre d'antécédents possibles |
| β : nombre d'antécédents effectifs |
| γ : valeur numérique associée à j |
| δ : coefficient sur j |

Les notes pour le critère α se répartissent entre 10 et 6 antécédents : le critère est peu pertinent. Pour ce premier choix, les notes β sont toutes nulles : β n'est pas pris en considération. Le critère suivant, γ , concerne l'optimisation : il n'a pas cours dans cette première phase. Le critère δ en revanche concerne un coefficient sur j qui apparaît dans l'ensemble des contraintes ; ce critère est pertinent.

Dans le but de favoriser la faisabilité, puisque le coefficient $p(j)$ intervient comme second membre d'une égalité, et pour laisser le plus de liberté possible aux i qui suivent, le programme va retenir le $p(j)$ le plus grand. Les premiers critères considérés par le programme pour le choix de i et j sont donc, en résumé : placer le papier le plus difficile à placer (critère e) sur la machine la moins chargée (critère *delta*). Le premier choix correspond donc à la fabrication du papier 3 sur la machine 3.

		1	2	3	4	5	6	7	8	9	10
		3	5	10	1	2	3	5	3	8	2
$p(j)$	\downarrow	9	13	9	1	3	14	15	7	9	15
		10	13	9	20	6	17	7	9	16	17
1	20	18	15	11	14	13	8	8	12	17	12
2	3	5	4	19	2	6	15	19	4	15	20

Choix 1 : $f(3) = 3$.

La capacité disponible de cette machine tombe à 10. Aucune autre implication n'apparaît. Pour le choix 2, les critères pertinents sont réexaminés tour à tour mais pour le programme, le critère e passe maintenant en tête devant a, car la distribution est devenue (8. 5. 5, 5...); le papier 9 est retenu. Les critères α et δ laissent alors les deux machines 2 et 3 à égalité : c'est finalement β qui les départage : dans la première phase, desserrer le plus possible les images est une bonne stratégie pour équilibrer à mesure les contraintes en vue de la faisabilité. Ici, la machine 2 n'est pas encore chargée, d'où le choix suivant.

Choix 2 : $f(9) = 2$.

La capacité disponible de la machine 2 devient alors 2, si bien que le programme, pour tenir compte de la contrainte $\sum q(i) = p(j)$ doit interdire $f(1)$, $f(2)$, $f(6)$, $f(7)$ et

$f(8)$ sur cette deuxième machine, il n'y a pas d'autre implication et le troisième choix conduit à reconsidérer en tête le critère a, puisque les deux qualités de papier - la 2 et la 7 - n'ont plus, à nouveau, que deux successeurs. Le critère e ne peut les départager puisque les demandes valent toutes deux 5 ; en dernier ressort c'est le critère f qui intervient : le programme opte pour celle des qualités dont la fabrication coûte, en tous cas, le plus cher. Le troisième choix effectué par le programme sera donc :

Choix 3 : $f(2) = 1$, pour un coût égal à 9.

La capacité résiduelle de la machine 1 passe donc à 4, si bien que le programme en déduit les Implications : $f(7) \neq 1$ et donc $f(7) = 3$. Le quatrième choix se fait suivant les mêmes critères. Le papier 6 doit être placé ; mais ici, en outre, le critère f laisse deux machines à égalité, puisque les coûts sont à 15, aussi bien sur la 1 que sur la 4. La machine 1, qui est la moins chargée est retenue (critère 6).

Choix 4 : $f(6) = 1$.

Comme il a été montré auparavant que les contraintes de capacité étaient de ce fait des égalités, et qu'un seul papier possède un $q(i)$ d'une unité qui est la capacité résiduelle de la machine 1, il vient maintenant directement : $f(4) = 1$. Donc, pour le papier 8, la machine la moins chère est la 4 d'où :

Choix 5 : $f(8) = 4$.

La machine 4 devient alors interdite, de telle sorte qu'obligatoirement : $f(1) = 3$. Pour les deux papiers restants, de demande égale à 2, le critère utile est le critère f, et permet de retenir :

Choix 6 : $f(5) = 3$.

D'où finalement, nécessairement : $f(10)=2$

La solution ainsi trouvée :

1	2	3	4	5	6	7	8	9	10
3	2	3	1	3	1	3	4	2	2

correspond à un coût total de 99.

Deuxième phase : Fort de cette solution réalisable, obtenue sans retour arrière, et après six choix seulement, le programme va être plus optimiste et tenter de reconstruire une « bonne » solution. Pour ce faire, il prend cette fois en considération les critères liés aux coûts, qu'il avait tout d'abord rejettés. Ainsi, si le papier retenu maintenant pour le premier choix est, pour les mêmes raisons, encore le n° 3, la machine sur laquelle le programme décide de le fabriquer est celle pour laquelle le coût effectif de fabrication est le plus petit (critère f).

Choix 1 : $f(3) = 3$.

Ce choix se trouve être le même que tout à l'heure, mais ensuite :

Choix 2 : $f(9) = 1$,

car c'est la première machine qui donne, pour cette qualité, une fabrication au moindre coût. Il vient de manière obligée : $f(4) = 1$ et le critère f provoque alors les choix successifs :

- et **Choix 3 :** $f(2) = 2,$
 Choix 4 : $f(7) = 3,$
 Choix 5 : $f(6) = 2,$
 et **Choix 6 :** $f(1) = 4,$
 Choix 7 : $f(8) = 3,$
 puis **Choix 8 :** $f(5) = 3,$
 d'où $f(10) = 2$

La nouvelle solution, obtenue avec ces critères d'optimisation, correspond à une valeur qui a été effectivement notablement améliorée :

1	2	3	4	5	6	7	8	9	10
4	2	3	1	3	2	3	3	1	2

Le coût total en est 84.

Troisième phase : Tout ceci n'est pas inutile, mais comment continuer pour avoir la solution optimale à coup sûr ? Continuons à suivre la démarche d'ALICE. L'expression de la fonction économique Z :

$$Z = \sum_{i \in Papiers} c(i, f(i))$$

montre, puisque chaque papier i doit recevoir une image $f(i)$, que la quantité Z est, en tous cas, supérieure à la somme des minimums des coefficients $c(i, j)$ pour chaque papier i . Soit dès lors \underline{Z} cette quantité :

$$\underline{Z} = \sum_{i \in Papiers} \min_{j \in Machines} c(i, j) \quad (8.3)$$

Nous avons vu que $\underline{Z} \leq Z$. Mais le programme effectue en outre une translation des coûts, en soustrayant effectivement à tous les coûts de fabrication du papier i (colonne i du tableau, ensemble des arcs values issus de i pour le programme) le coût minimum pour ce papier. Les nouveaux coûts $c'(i, j)$ sont alors les suivants :

	1	2	3	4	5	6	7	8	9	10
	3	5	10	1	2	3	5	3	8	2
1	9	8	0		1	8	8	0	5	0
2	10	8	0	9	4	11	0	2	12	2
3	20	13	6	0	12	7	1	1	8	2
4	3	0			0	0	8		0	19
Quantités retranchées	5	9	1	2	6	7	7	4	15	1

avec somme des quantités retranchées $= \underline{Z} = 67$ et la relation exacte :

$$Z = \underline{Z} + \sum_i c'(i, j) \quad (8.4)$$

Par ailleurs la meilleure solution connue vaut $Z_0 = 84$ et, puisque les $c'(i, j)$ sont positifs par construction, on a :

$$\begin{aligned} \underline{Z} &\leq Z^* \leq Z_0 \\ \text{soit : } &67 \leq Z^* \leq 84 \end{aligned}$$

où Z^* représente la valeur optimum cherchée.

ALICE élimine alors tous les coûts supérieurs strictement à $84 - 67 - 1 = 16$ qui conduiraient nécessairement à une solution moins bonne que celle obtenue en phase 2. Un arc

est dans ce cas : l'arc (10,4) de coût réduit 19. Aucune autre implication n'est disponible et le premier choix d'ALICE se fait maintenant selon le critère d'optimisation - g - du regret maximum : l'arc de coût nul, qui entraînerait le coût le plus élevé s'il était éliminé, est choisi. Ces regrets sont ici simplement égaux aux second coûts minimum sur chaque colonne. Ils valent :

Papier	1	2	3	4	5	6	7	8	9	10
Regret	8	0	9	1	7	1	1	5	2	7

De telle sorte que le premier choix de cette dernière phase est à nouveau (même choix pour trois critères différents depuis le début !) :

Choix 1 : $f(3) = 3.$

Il n'est suivi d'aucune implication. Le même critère donne alors :

Choix 2 : $f(1) = 4.$

Ceci entraîne $f(i) \neq 4$ pour tous les autres papiers. En particulier, les coûts réduits nuls pour les papiers 4, 5 et 8 deviennent interdits. ALICE soustrait donc les nouveaux minimum sur ces colonnes, soit 1, 7 et 5, pour faire réapparaître sous les deux hypothèses faites, de nouveaux coûts réduits et une borne inférieure \underline{Z} (Choix 1, Choix 2) qui vaut :

$$Z(\text{Choix 1, Choix 2}) = Z + 1 + 7 + 5 = 80.$$

Dans ce contexte, tous les coûts réduits supérieurs strictement à 3 ne peuvent donner une solution qui intéresse le programme ; ils sont éliminés. Il reste donc

	1	2	3	4	5	6	7	8	9	10
1		5		1	2	3	5	3	8	2
2	9		0		0	1		0	0	0
3	10		0		3		0	2		0
					0	1	1	3	2	
						-1	-7		-5	

Ces éliminations entraînent aussitôt : $f(10) = 2$, et le troisième choix se fait de la même façon :

Choix 3 : $f(8) = 1.$

Il implique $f(9) \neq 1$ par la contrainte de capacité. Ainsi, une nouvelle soustraction est rendue nécessaire pour ce papier. \underline{Z} devient :

$$\underline{Z} (\text{choix 1, choix 2, choix 3}) = \underline{Z} (\text{choix 1, choix 2}) + 2 = 82$$

si bien que seuls les coûts nuls ou unités sont autorisés ; en particulier :

$$f(4) * 2 \text{ donc } f(4) = 1$$

Puis le **Choix 4 :** $f(7) = 1$
 entraîne : $f(2) \neq 1,$
 et : $f(5) \neq 1,$
 d'où : $f(5) = 3,$
 et : $f(2) = 2.$

Ces affectations impliquent à leur tour :

$f(9) \neq 2$
 et : $f(6) \neq 3$
 d'où : $f(9) = 3,$
 et : $f(6) = 2.$

qui constitue bien, finalement, une **solution de coût 82.**

- **Preuve de l'optimalité :**

Il ne reste à ALICE qu'à revenir sur ces quatre choix, un à un, pour prouver l'optimalité, ou exhiber une solution meilleure encore.

Choix 4 : $f(7) = 1$. Ce retour de choix est trivial : connaissant maintenant l'existence d'une solution de coût 82, seul un regret nul conviendrait : $f(7) = 1$ est en fait une implication, puisque son regret vaut 1.

Choix 3 : $f(7) = 1$. Nous avons ici une borne supérieure valant 80 et le regret du choix $f(8) = 1$ était 3 : à nouveau, ce choix se transforme, lors de la remontée, en une implication qui provoque un nouveau retour arrière.

Choix 2 : $f(1) = 4$. La borne supérieure vaut 67 et le regret associé vaut 8. ALICE étudie donc le cas $f(1) \neq 4$. Cette suppression fait passer la quantité \underline{Z} (choix 1) à la valeur $67 + 8 = 75$. Par ailleurs, puisque trois autres possibilités sont valides pour placer le papier 1, ALICE, réétudie la situation : les arcs (10, 1) et (10, 3) sont supprimés, ce qui implique $f(10) = 2$; de la même façon (4, 1), (5, 2), (5, 3) sont supprimés et $f(5) = 4$.

La contrainte de capacité impose alors $f(4) = 4$. La quatrième machine est ainsi saturée, donc : $f(8) \neq 4$ mais comme, pour satisfaire la contrainte de coût, on a :

$f(8) \neq 2$
 et : $f(8) \neq 3,$
 il vient : $f(8) = 1,$
 avec : $c(8, 1) = 5,$
 soit : $\underline{Z}(\text{choix 1}) = 75 + 5 = 80.$

Il ne faut donc compléter la solution qu'avec les seuls arcs de coût inférieur à 2. Or $f(9) \neq 1$, puisque la capacité résiduelle de la première machine est insuffisante pour ce papier. Il est ainsi impossible de compléter la solution précédente. Il ne reste donc plus que le premier choix.

Choix 1 : $f(3) = 3$.

La seule possibilité est de faire $f(3) = 2$ avec $Z = 67 + 9 = 76$. Tous les coûts supérieurs à 5 = 82 - 76 - 1 sont supprimés. En particulier, il faut faire à la fois $f(1) = 4$ et $f(5) = 4$. Ces deux affectations, représentant une quantité globale de 5 unités sur une machine de capacité 3 sont incompatibles. Il est donc ici impossible d'obtenir une solution de coût inférieur à 82. L'arborescence de recherche est finalement celle de la figure 8.11 :

ALICE a ainsi démontré l'optimalité de la solution obtenue

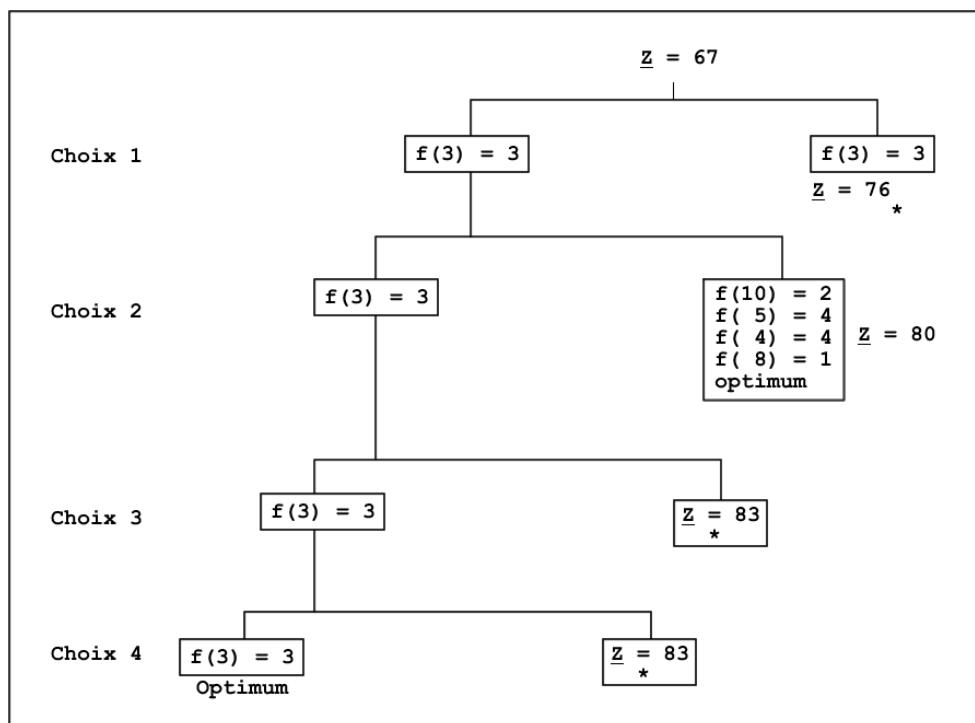


FIG. 8.11 – Solution optimale de coût 82

La fabrication :

$$\begin{array}{ll} \text{PAPIER} & [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10] \\ \text{MACHINE} & [4 \ 2 \ 3 \ 1 \ 3 \ 2 \ 1 \ 1 \ 3 \ 2] \end{array}$$

a pour coût 82 et est minimale.

8.7 Table générale des problèmes traités par ALICE :

- a : Nombre d'inconnues (sommets de D) ;
- b : Nombre de valeurs possibles (sommets de A) ;
- c : Nombre de contraintes algébriques ;
- d : Temps de résolution en secondes d'IBM 43 31 ;
- e : Nombre de retours arrière.

	a	b	c	d	e
Exemple 1 : ordonnancement	9	9	1	1	0
Exemple 1 : modifié (Cf. 2.3)	9	9	1	2	1
Exemple 2 : inéquations	5	2	3	1	0
Exemple 3 : localisation	5	8	1	2	0
Génération de permutations	9	9	33	4	0
Puzzle logique (Cf. 7.2)	25	25	14	4	1
Emploi du temps	11	11	6	2	0
Flot arbre	10	10	1	2	0

Recouvrement	5	8	2	3	0
Huit reines	8	8	1	5	0
Cinq reines	64	64	1	1	0
Instant insanity	8	12	20	11	3
informations fausses	10	10	8	1	0
Mike & John au tennis	19	6	21	3	1
Send + more = money	12	18	7	4	0
Crypt addition paramétrée	14	12	9	3	0
Donald + Gerald = Robert	15	12	9	9	4
Lyndon * b = Johnson	16	14	7	4	0
Nombres croisés	26	10	28	6	0
Opérations croisées	21	20	37	7	0
Multiplication	29	20	23	249	1826
Carré magique d'ordre 3	9	9	8	4	0
Carré magique d'ordre 4	16	16	10	7	0
Carré hétérogène	17	29	1	10	7
Cube d'ordre 3	27	27	50	18	0
Sphère magique	26	8	6	0	
Disques	4	8	9	3	0
Missionnaires & Cannibales	80	13	10	11	4
Cavalier d'Euler	64	64	0	5	0
Picnic	9	6	7	4	0
Couple au tennis	18	18	47	3	0
Allocation de ressources	6	3	2	1	0
Partitionnement	15	32	1	4	4
	32	327	1	47	128
	43	327	1	86	628
	45	512	1	91	815
	130	878	1	186	2464
Recouvrement	15	32	1	4	4
	130	878	1	75	429
Recouvrement Bottle-Neck	45	512	1	8	0
Découpe linéaire	9	9	2	2	1
	54	54	2	3	0
	53	53	2	5	7
	647	70	2	41	953
Choix d'investissement	60	2	30	46	0
	60	2	30	49	0
Programmes linéaires	8	2	4	3	1
	10	2	6	2	0
Test Haldi IBM	15	35	1	120	2824
Programmes quadratiques	6	2	4	4	4
	18	2	4	29	37
	24	2	4	39	98
	32	2	4	92	411
	40	2	4	309	1677
Ordonnancement	100	250	300	26	0
Ordonnancement disjonctif	100	250	310	54	27
Plus court chemin	25	25	56	17	14
Flot maximum	15	65	28	15	16
Affectation	20	20	0	10	18

Transport	60	60	0	6	0
Arbre minimum	110	110	0	5	0
Couplage maximum	60	60	0	9	5
Parcours Eulerien	60	60	0	9	5
Découpe dans le plan	24	20	14	4	0
	30	28	22	7	0
Agencement d'un plan au sol	36	26	107	17	0
Voyageur de commerce	6	6	1	1	0
	10	10	1	6	27
	25	25	1	47	433
Existence d'un circuit Hamiltonien	10	10	0	1	0
	20	20	0	2	0
	64	64	0	5	0
	110	110	0	73	1220
	150	150	0	188	2636
Tournées avec contraintes	10	10	2	13	18
	25	25	2	67	625
Coloration	40	40	1	17	1
	99	99	1	71	124
	150	130	1	162	250
	41	4	1	6	0
	110	4	1	12	4
	150	4	1	18	24
Planning	186	27	2	16	12
	186	28	2	4	0
CouplageMaximal	12	2	2	2	1
	32	2	2	2	0
	80	2	2	241	2056
Papeterie	4	10	2	6	1
	1030	2	17	14	
Localisation	10	10	2	5	0
	40	40	2	17	24
Roulements SNCF	40	40	2	17	24
	40	40	2	14	47
	53	53	2	31	138

8.8 Généralité et efficacité

Il a déjà été dit plus haut qu'ALICE était efficace et pouvait même conduire à de meilleures performances que de bons programmes spécifiques. Ce résultat peut paraître surprenant et nous l'expliquerons à travers des exemples. Deux raisons essentielles sont à la base de ce succès : la première est la force de la **manipulation formelle des expressions**, par rapport aux seuls calculs numériques habituellement entrepris en résolution de problèmes : la propagation symbolique des contraintes permet de retarder les choix le plus longtemps possible. La seconde tient à la qualité des choix, quand ceux-ci deviennent nécessaires : la batterie des critères possibles est très complète, puisque prévue pour toute une famille de problèmes. Qu'un cas imprévu vienne à se présenter, et le schéma d'énumération prévu par le programmeur d'un algorithme spécifique aura un très mauvais comportement.

ALICE, par contre, étudie soigneusement le contexte et recherche, non seulement pour chaque problème, mais aussi pour chaque choix à l'intérieur de ce problème, un jeu de critères pertinent. Dans le pire des cas, son comportement sera le comportement figé du programme spécifique. La plupart du temps, ALICE considérera, pour certains choix au moins, un critère inattendu, a priori prévu pour une toute autre famille de problèmes, et en réalité tout à fait adapté à la situation. C'est ainsi que l'on constate pour certains problèmes solubles avec papier et crayon que la démarche d'Alice conduit déjà à des arbres de recherche plus petits que ceux engendrés à la main. Les paragraphes qui suivent le montrent.

8.8.1 Exemple 1

Les problèmes de cryptarithmétique sont classiques, n'exigent pas de grandes connaissances mathématiques, et à ce titre, constituent un bon exemple de comparaison. Une opération simple est posée, par exemple une addition. Les chiffres ont été remplacés par des lettres. On suppose qu'à chaque lettre correspond un chiffre (en base 10) et un seul, et qu'à deux lettres différentes correspondent deux chiffres différents. Il s'agit de retrouver les valeurs des lettres qui rendent l'opération correcte. Un programme énumératif et numérique peut être écrit pour résoudre ce genre de problèmes par un schéma classique (Cf chapitre V). Les valeurs des lettres opérées peuvent être choisies de droite à gauche pour avoir facilement les retenues, et l'opération effectuée au fur et à mesure, pour tester la validité des valeurs en cours. Le schéma général est :

```

DEBUT
j ← 1
CHOIX (injectif) des valeurs des lettres de la j-ième colonne.
Calcul du j-ième résultat.
Test de faisabilité
SI le résultat est différent des valeurs déjà choisies
ALORS    progresser ;
SINON    revenir sur le dernier choix.
FSI
FSI

```

Ce programme sera un premier élément de référence. Newell et Simon (1978) ont par ailleurs étudié longuement la façon humaine de procéder sur ce type d'exercice. Ils ont, en particulier, étudié les protocoles de plusieurs sujets et se sont intéressés à un modèle, par système de productions, des procédures de résolution. Un exemple intensivement testé est :

$$\begin{array}{r}
 \text{GERALD} \\
 +\text{DONALD} \\
 \hline
 = \text{ROBERT}
 \end{array}$$

Les lettres de gauche : G, D et R ne sont pas nulles. Les solutions comportent plusieurs pages, alors même que la valeur de D est donnée aux sujets. Exercez votre habileté sur cet énoncé, puis nous comparerons avec la solution d'ALICE. SI vous décrivez tout l'espace de recherche en ayant toutes les solutions en moins d'une vingtaine de noeuds, vous battez de loin les meilleurs sujets étudiés par Newell et Simon.

Remarquons sur cet exemple le handicap fondamental de l'algorithme énumératif aveugle. Seule une faible partie de l'information est propagée à chaque choix, et dans un ordre inadapté. L'ordre convenable dépend en effet de chaque opération. Il ne peut

être trouvé qu'en utilisant précisément un traitement symbolique des contraintes. Le nombre d'essais à effectuer dépend du nombre de lettres-opérandes indépendantes. qui Ici sont au nombre de six : D, L, A, N, O et G. C'est donc environ 101/41, soit 151. 200 essais qu'il faudra tenter. Par contre. ALICE résout complètement le problème en exactement six noeuds terminaux.

Donnons-nous les symboles a, b, c, d et e pour les retenues, prises de gauche à droite. Les contraintes initiales s'écrivent :

(1)	$2 \cdot D = T + 10 \cdot e$
(2)	$e + 2 \cdot L = R + 10 \cdot d$
(3)	$d + 2 \cdot A = E + 10 \cdot c$
(4)	$c + R + N = B + 10 \cdot b$
(5)	$b + E + O = O + 10 \cdot a$
(6)	$a + G + D = R$

Les lettres doivent être en bijection avec l'intervalle entier $\llbracket 0, 9 \rrbracket$. Les lettres G, D et R sont différentes de 0.

Premières inférences (dans l'ordre trouvé par le système)

T est pair par (1).

$(a = 0 \wedge b = 0 \wedge E = 0)$ OU $(a = 1 \wedge b = 1 \wedge E = 9)$ par (5).

$R \equiv e$ (modulo 2) par (2). $E \equiv d$ (modulo 2) par (3).

En outre, toutes les retenues sont au plus égales à 1, car la somme de deux lettres ne peut dépasser 17.

Aucune autre information n'est déduite et le premier choix doit être fait ; la contrainte OU qui vient d'être engendrée fournit un bon support.

Choix 1 : $E = 0$ et donc $a = b = 0$.

Trois contraintes sont changées et réétudiées, tandis que (5) est devenue trivialement satisfaites.

(3i)	$d + 2 \cdot A = 10 \cdot c$
(4i)	$c + R + N = B$
(6i)	$G + D = R$

Déductions : d est pair par (3i) donc $d = 0$ et $A = 5 \cdot c$.

L'image 0 est déjà atteinte par E.

Le graphe répond donc $A = 5$.

Ainsi $c = 1$ et (3i) est supprimée.

La contrainte (6i) donne : $R > 3$ car G et D doivent être différentes, et $G + D < 9$. d'où $G < 8$. et $D < 8$. En outre. $G + D$ est substitué à R dans (4i). De (4i) le modulé arithmétique (décrit en 4.3.2) déduit deux nouvelles contraintes :

$$(7i) 1 + G + D + N \leq 9 \text{ (valeur supérieure de B) et } B \geq 7.$$

puisque le plus petit jeu de trois valeurs différentes dans l'espace-image donne une somme égale à 6. L'inégalité (7i) donne à son tour (type spécial 5.2.2) :

$$G \leq 5, D \leq 5, N \leq 5 \text{ et aussi } G + D = R \leq 7.$$

L'étude des contraintes (1), (2) et (6i) est relancée, puisque les valeurs de leurs inconnues ont été modifiées. Comme $R \leq 7$. (6i) entraîne maintenant $G \leq 6$ et $D \leq 6$. Alors que (1) ne donne rien de nouveau, (2) entraîne : $L \leq 3$, puisque $R \leq 7$; $L \geq 1$, puisque $R \leq 3$.

Toutes les contraintes ont été étudiées et un nouveau choix est nécessaire. Plusieurs variables sont candidates (B et les retenues), avec seulement deux valeurs possibles. C'est le critère de difficulté qui les départage. L'intérêt de e, qui figure dans les deux contraintes simples, est le plus grand.

Choix 1.1 : $e = 0$. Réécrivons le système des contraintes :

$$\begin{array}{ll} (1\text{iii}) & 2 \cdot D = T \\ (2\text{iii}) & 2 \cdot L = R \\ (4\text{iii}) & 1 + G + D + N = B \\ (6\text{iii}) & G + D = R \end{array} \quad \left. \begin{array}{l} \text{contraintes portées par le graphe} \\ \text{avec } T \text{ et } R \text{ pairs} \end{array} \right\}$$

Inférences. Le graphe « sait » qu'en outre :

$$4 \leq B \leq -9, 1 \leq N \leq 5, 3 \leq R \leq 7, 1 \leq D \leq 5, 1 \leq L \leq 3, 1 \leq G \leq 5.$$

L'analyse de nouvelles contraintes (1ii) et (2ii) affine immédiatement ces bornes :

$$\begin{aligned} L &= 2 \text{ ou } 3, \\ R &= 4 \text{ ou } 5. \end{aligned}$$

ALICE termine ici par énumération globale, à partir de la contrainte (2ii), la plus simple et dans laquelle les variables ont le plus petit espace-image. Si $R = 4$ alors $L = 2$, $D + G = 4$ entraîne $D \geq 3$, donc $T = 6$ car les valeurs 0, 2, 4 et 8 sont interdites. Ainsi $D = 3$ et $G = 1$; mais alors tous les points 0, 1, 2, 3, 4 et 5 sont pris et (4ii) $B = 5 + N$ ne peut donner de solution. Si $R = 6$ alors $L = 3$ et $\{G,D\} = \{2,4\}$ donc $T = 8$ puisque la valeur 4 est déjà prise. Ainsi M vient $D = 4$ et $G = 1$. A nouveau, il n'y a plus de valeurs possibles pour N dans l'équation (4ii) :

$$B = 7 + N.$$

Le dernier choix est donc remis en cause. Il ne reste que :

Choix 1.2 : $e = 1$. Outre les équations (4ii) et (6ii), on a maintenant :

$$\begin{array}{ll} (1\text{iii}) & 2 \cdot D = T + 10 \\ (2\text{iii}) & 1 + 2 \cdot L = R \end{array}$$

R doit donc être impair. Or le contexte du choix 1, qui a été rétabli, indique :

$$3 \leq R \leq 7$$

La valeur 5 est déjà prise par A, donc $R = 3$ ou 7. Mais l'analyse de (1iii) donne $D \geq 6$. Donc-par (6ii), on a : $R \geq 7$.

Finalement, $R = 7$, avec $L = 3$ et $B = N + 8$, soit $N = 1$ et $B = 9$. Il reste :

$$D + G = 7. \text{ Or } D \geq 6 \text{ et } G \geq 1$$

et la valeur 1 est déjà prise par N, d'où une nouvelle impossibilité. Le tout premier choix doit donc être remis en question.

Choix 2. $E = 9$, etd onc $a = b = 1$. D'où, immédiatement, les nouvelles équations :

$$\begin{array}{ll} (3\text{iv}) & d + 2 \cdot A = 9 + 10 - c \\ (4\text{iv}) & c + R + N = B + 10 \\ (6\text{iv}) & 1 + G + D = R \end{array}$$

Ensuite, la contrainte (3iv), étudiée la première, implique par parité : $d = 1$. d'où :

$$(2\text{iv}) \quad e + 2 - L = R + 10$$

avec $A = 4$, car E vaut déjà 9 et $c = 0$ donc :

$$(4v) \quad R + N = B + 10$$

L'analyse de (2iv) entraîne, avec $L \leq 8$ et $R \geq 4$, d'une part :

$$\begin{array}{ll} 17 \geq R + 10, & \text{soit } R \leq 7 \\ \text{d'autre part :} & e + 2 \cdot L \geq 14, \text{ soit } L \geq 7, \text{ donc } L = 7 \text{ ou } 8. \end{array}$$

Les autres contraintes permettent on outre d'inférer $B \leq 5$ et $N \geq 3$. Mais un autre choix est nécessaire. L , qui n'a plus que deux successeurs, en donne le support :

Choix 2.1 : $L = 7$. La première contrainte étudiée est :

$$(2v) \quad e + 14 = R + 10$$

Mais A vaut 4, on a donc obligatoirement $R = 5$ et $e = 1$. Portant cette information dans (4v) et (6iv). Il vient : $N = B + 5$ et $D + G = 4$. On a donc : $D \leq 3$ et $G \leq 3$. Par ailleurs la contrainte :

$$(1v) \quad 2 * D = T + 10$$

implique : $D \geq 5$, ce qui provoque une contradiction, il reste le dernier cas :

Choix 2.2 : $L = 8$. Ici on a :

$$\begin{array}{lll} (2vi) & e + 6 & = R \\ (4vi) & R + N & = B + 10 \\ (6vl) & 1 + G + D & = R \end{array}$$

On doit alors avoir simultanément, par (2vi) : $R = 6$ ou 7 et aussi : $G + D = 5 + e$ par substitution ; d'où ALICE déduit : $G \leq 5$, $D \leq 5$. Une énumération à partir de (2vi) va donner :

<p>1. Si $R = 7$ alors :</p> <table style="margin-left: 40px;"> <tr><td>e</td><td>=</td><td>1</td></tr> <tr><td>$G + D$</td><td>=</td><td>6</td></tr> <tr><td>$2 \cdot D$</td><td>=</td><td>$T + 10$</td></tr> <tr><td>donc</td><td>D</td><td>= 5</td></tr> <tr><td>et</td><td>T</td><td>= 0</td></tr> <tr><td>avec</td><td>G</td><td>= 1</td></tr> <tr><td>et</td><td>N</td><td>= $B + 3$</td></tr> <tr><td>puis</td><td>B</td><td>= 3</td></tr> <tr><td>et</td><td>N</td><td>= 6</td></tr> </table> <p>soit la solution :</p> $\begin{array}{ccccccccc} < & G & E & R & A & L & D & O & N & B & T > \\ < & 1 & 9 & 7 & 4 & 8 & 5 & 2 & 6 & 3 & 0 > \end{array}$	e	=	1	$G + D$	=	6	$2 \cdot D$	=	$T + 10$	donc	D	= 5	et	T	= 0	avec	G	= 1	et	N	= $B + 3$	puis	B	= 3	et	N	= 6	<p>2. Si $R = 6$, alors :</p> <table style="margin-left: 40px;"> <tr><td>e</td><td>=</td><td>0</td></tr> <tr><td>$G + D$</td><td>=</td><td>5</td></tr> <tr><td>$2 \cdot D$</td><td>=</td><td>T</td></tr> <tr><td>donc</td><td>D</td><td>= 1</td></tr> <tr><td>et</td><td>T</td><td>= 2</td></tr> <tr><td colspan="3">car les valeurs 4 et 6 sont prises.</td></tr> <tr><td colspan="3">Mais alors $G = 5 - D$ vaut 4 qui est déjà valeur de A, d'où une dernière impossibilité</td></tr> </table>	e	=	0	$G + D$	=	5	$2 \cdot D$	=	T	donc	D	= 1	et	T	= 2	car les valeurs 4 et 6 sont prises.			Mais alors $G = 5 - D$ vaut 4 qui est déjà valeur de A , d'où une dernière impossibilité		
e	=	1																																															
$G + D$	=	6																																															
$2 \cdot D$	=	$T + 10$																																															
donc	D	= 5																																															
et	T	= 0																																															
avec	G	= 1																																															
et	N	= $B + 3$																																															
puis	B	= 3																																															
et	N	= 6																																															
e	=	0																																															
$G + D$	=	5																																															
$2 \cdot D$	=	T																																															
donc	D	= 1																																															
et	T	= 2																																															
car les valeurs 4 et 6 sont prises.																																																	
Mais alors $G = 5 - D$ vaut 4 qui est déjà valeur de A , d'où une dernière impossibilité																																																	

La solution est donc unique, et l'arbre de recherche est représenté par la figure 8.12. Cet arbre possède six feuilles, complètement développées en 9 secondes, soit moins de temps qu'il n'en faut au programme énumératif du type donné au chapitre V (1 minute et 10 secondes sur IBM 43 31).

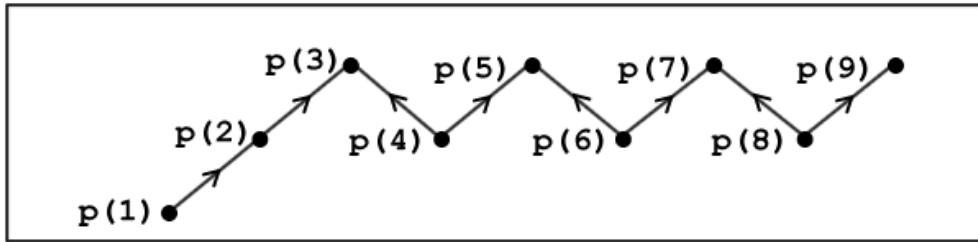


FIG. 8.12 – Arbre de recherche d'ALICE pour GERALD + DONALD = ROBERT.
Une feuille marquée * correspond à une impossibilité.

8.8.2 Exemple 2

Prenons un problème plus compliqué. L'énoncé suivant, qui traite de construction de permutations a été soumis par Marcel P. Schutzenberger.

« Trouver toutes les permutations p sur les n premiers entiers qui répondent à deux contraintes définies à partir de deux vecteurs, m et a , sur l'intervalle $\llbracket 1, n-l \rrbracket$.

Le vecteur de monotonies, $m : \llbracket 1, n-l \rrbracket \leftarrow \{0, 1\}$ impose :

$\{p(i+1) \text{ est plus grand que } p(i) \text{ ssi } m(i) = 1\}$ soit formellement :

$$\forall i, i \in \llbracket 1, n-l \rrbracket, m(i) = 1 \Leftrightarrow p(i+1) > p(i)$$

Le vecteur d'avances, $a : \llbracket 1, n-l \rrbracket$ impose quant à lui :

si $a(i) = 0$ alors $p(i) + 1$ est à gauche de $p(i)$;

si $a(i) = 1$ alors $p(i) + 1$ est à droite de $p(i)$;

et de plus $p(i) \neq n$; soit formellement :

$$\forall i, i \in \llbracket 1, n-l \rrbracket$$

$$a(i) = 0 \Rightarrow \forall j, j \in \llbracket i+1, n \rrbracket \quad p(j) \neq p(i) + 1 \quad (1)$$

$$a(i) = 1 \Rightarrow (\forall j, j \in \llbracket 1, i-1 \rrbracket \quad p(j) \neq p(i) + 1) \text{ ET } p(i) \neq n \quad (2)$$

(8.5)

Le vecteur m indique donc le sens de variation de la permutation d'un élément au suivant, tandis que le vecteur a indique que l'élément $p(i) + 1$ doit se trouver soit avant, soit après l'élément $p(i)$ dans la permutation à chercher.

Premier exemple numérique simple :

Si les données sont : $m \equiv [0 \ 1 \ 1]$ et $a \equiv [1 \ 1 \ 0]$, on a :

par $m(1) = 0$:	$p(2) < p(1)$	(a)
par $m(2) = 1$:	$p(3) > p(2)$	(b)
par $m(3) = 1$:	$p(4) > p(3)$	(c)
donc :	$p(4) > p(3) > p(2)$	
soit :	$p(4) \geq 3$	(d)
	$p(3) \geq 2$	(e)
	$p(3) \leq 3$	(f)

Or, à ce stade, la contrainte (a) impose :

$$p(1) \geq 2 \quad (g)$$

et seul $p(2)$ peut prendre la valeur 1 :

$$p(2) = 1 \quad (h)$$

Par ailleurs les contraintes issues du vecteur a donnent :

par $a(1) = 1$	$p(1) \neq 4$	(i)
par $a(2) = 1$	$p(1) \neq p(2) + 1$	
soit :	$p(1) \neq 2$	(j)
et :	$p(2) \neq 4$ (déjà satisfaite)	
par $a(3) = 0$	$p(4) \neq p(3) + 1$	

Par les contraintes (f) et (i), un seul antécédent est possible pour 4 :

$$p(4) = 4.$$

Et la contrainte (j) impose à son tour :

$$p(1) = 3,$$

$$\text{d'où :} \quad p(3) = 2,$$

tandis que toutes les contraintes sont satisfaites.

Une seule permutation convient, donc, finalement :

$$p \equiv [3 \ 1 \ 2 \ 4]$$

ALICE fait ici exactement le même raisonnement que celui que nous venons d'effectuer à la main. Si, en revanche, nous voulons écrire un programme spécifique pour ce problème, le script « énumération » se déclenche immédiatement. Il comporte trois phases principales et se résume, comme d'habitude, dans le schéma :

A : Placer un objet à la fois.

B : Tester la solution partielle et progresser si succès.

C : Revenir en arrière en cas d'échec partiel, ou si une solution a pu être atteinte

Le point A correspond au placement de l'élément suivant. En prenant l'ordre « naturel » sur $\llbracket 1, n \rrbracket$ on fera :

$$\text{A : } i \leftarrow i + 1 \text{ ET } p(i) \leftarrow p(i) + 1$$

à partir des valeurs initiales nulles.

Le point B est le point délicat : l'ordre dans lequel seront effectués les différents tests détermine l'efficacité de notre programme. Il faut prévoir trois tests :

$$\begin{cases} t_1 : p \text{ est-elle une permutation ?} \\ t_2 : \text{les contraintes de monotonie sont-elles satisfaites ?} \\ t_3 : \text{les contraintes d'avance sont-elles satisfaites ?} \end{cases}$$

Pour éviter tout essai inutile, ces tests doivent être placés le plus tôt possible. Le test t_1 peut être entrepris dès la génération des $p(i)$ et doit donc être effectué avant t_3 . Le test t_3 pose un problème lorsque $a(i) = 0$. Les éléments de p , d'indices supérieurs à i sont encore inconnus. Mais un résultat élémentaire vient à l'esprit : puisque p est une permutation, donc une bijection, dire que tous les $p(j)$ sont différents de $p(i) + 1$ pour $j \in \llbracket i + 1, n \rrbracket$ revient à dire que, soit $p(i) + 1$ est rencontré avant i , donc dans

les éléments déjà placés, soit $p(i)$ vaut n . Le test t_3 peut donc s'inscrire dans notre algorithme.

Le test t_2 est, par contre, très simple. Il élimine a priori, en moyenne, plus de cas que le test t_1 . Nous le mettrons donc en tête. Nous pouvons maintenant préciser notre schéma de programme.

Procédure PERM :	Engendre toutes les permutations des n premiers entiers selon les vecteurs booléens a et m donnés.
TANT QUE $i > 0$	
$I \leftarrow$ ensemble des entiers de 1 à $i - 1$; {Tester les monotonies pour voir si $p(i)$ peut augmenter :}	
TANT QUE $(m(i - 1) = 1 \text{ ET } p(i) < n)$ OU $(m(i - 1) = 0 \text{ ET } p(i) < p(i - 1) - 1)$	
$p(i) \leftarrow p(i) + 1$;	
{Test permutation}	
SI $\forall j \in I p(j) \neq p(i)$	
ALORS	
{tester les avances}	
$q \leftarrow p(i) + 1$; $ok \leftarrow$ faux	
SI $a(i) = 0$ { q présent dans I ou $p(i) = n$ }	
ALORS SI $(p(i) = n) \text{ OU } (\exists j \in I p(j) = q)$	
ALORS $ok \leftarrow$ vrai	
FSI	
SINON { $\{a(i) = 1, q$ absent dans I et $p(i) \neq n\}$	
SI $(p(i) \neq n) \text{ ET } (\forall j \in I p(j) \neq q)$	
ALORS $ok \leftarrow$ vrai;	
FSI	
FSI	
SI ok {on progresse sur i }	
ALORS SI $i = n$	
ALORS imprimer la solution : $p(i) \leftarrow n$; {on force le retour arrière}	
SINON $i \leftarrow i + 1$	
FSI	
FSI {retour dans tous les cas d'échec}	
FTQ {retour arrière lorsque $p(i)$ ne peut plus être augmenté}	
$p(i) \leftarrow 0$; $i \leftarrow i - 1$;	
FTQ	
FIN PERM	

Ce programme peut être amélioré en repérant encore quelques cas spéciaux. Cependant, une remarque s'impose : ce programme est **figé**. Il a été écrit, comme tout algorithme qui se respecte, **sans tenir compte des données numériques...** Les permutations seront toutes engendrées dans le même ordre, quels que soient les vecteurs a et m . Ceci peut être très dangereux puisque certaines valeurs des $p(i)$ peuvent être imposées par les données. Et ces valeurs peuvent très bien se trouver dans les dernières, le plus à droite. Que d'essais inutiles alors !

Rien n'exclut, même dans ce problème dont la solution est a priori exponentielle par énumération, que des données particulières contraignent suffisamment la permutation

p pour que la solution soit trouvée en un temps parfaitement polynômial.

8.8.3 Exécution en ALICE

De tout cela, ALICE est à même de se rendre compte. Tout d'abord, loin de bâtir un algorithme unique, elle travaille avec les données. Supposons, par exemple, que $n = 9$ et que a et m soient les vecteurs :

$$\begin{aligned} a &= (1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0) \\ m &= (1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1) \end{aligned}$$

Les contraintes sur m engendrent, lorsque $m(i) = 0$:

$$\begin{aligned} &\neq (p(i+1) > p(i)), \\ \text{soit : } &p(i+1) \leq p(i) \\ \text{Mais } p \text{ est une bijection, donc } &p(i+1) < p(i), \\ \text{Il vient ainsi : } &p(2) > p(1), \\ &p(3) > p(2), \\ &p(4) < p(3), \\ &p(5) > p(4), \\ &p(6) < p(5), \\ &p(7) > p(6), \\ &p(8) < p(7), \\ &p(9) > p(8). \end{aligned}$$

Les arcs $(i,9)$ sont supprimés dans le graphe pour tous les i tels que $a(i) = 1$. Les contraintes sur a engendrent, quant à elles, 22 inéquations.

$$\begin{aligned} a(2) = 1 &\Rightarrow \\ p(1) &\neq p(2) + 1 ; \\ a(3) = 1 &\Rightarrow \\ p(1) &\neq p(3) + 1 ; p(2) \neq p(3) + 1 ; \\ a(4) = 1 &\Rightarrow \\ p(1) &\neq p(4) + 1 ; p(2) \neq p(4) + 1 ; p(3) \neq p(4) + 1 ; \\ a(5) = 0 &\Rightarrow \\ p(6) &\neq p(5) + 1 ; p(7) \neq p(5) + 1 ; p(8) \neq p(5) + 1 ; p(9) \neq p(5) + 1 ; \\ a(6) = 1 &\Rightarrow \\ p(1) &\neq p(6) + 1 ; p(2) \neq p(6) + 1 ; p(3) \neq p(6) + 1 ; p(4) \neq p(6) + 1 ; \\ p(5) &\neq p(6) + 1 ; \\ a(7) = 1 &\Rightarrow \\ p(1) &\neq p(7) + 1 ; p(2) \neq p(7) + 1 ; p(3) \neq p(7) + 1 ; p(4) \neq p(7) + 1 ; \\ p(5) &\neq p(7) + 1 ; p(6) \neq p(7) + 1 ; \\ a(8) = 0 &\Rightarrow \\ p(9) &\neq p(8) + 1 ; \end{aligned}$$

Les contraintes en \leq sont préférées par le système aux contraintes en \neq . Ces contraintes sont en effet étudiées dans un ordre qu'ALICE sait être convenable : il est donné par calcul d'une fonction de Grundy (Cf. paragraphe 5.2.3) qui les emboîte dans le bon sens. Cela revient à faire en machine l'analogie du petit dessin de la figure 8.13.

La première contrainte examinée est : $p(2) > p(1)$ d'où $p(2) \geq 2$. Alors :

$$p(3) > p(2) \text{ entraîne } p(3) \geq 3.$$

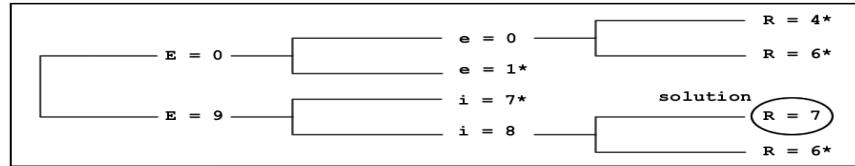


FIG. 8.13 – Graphe des contraintes de monotonies

De même $p(5) \geq 2$; $p(7) \geq 2$; $p(9) \geq 2$ et $p(4) \leq 8$; $p(6) \leq 8$; $p(8) \leq 8$; $p(2) \leq 8$, et $p(1) \leq 7$.

Certaines contraintes en \neq , qui sont maintenant étudiées, sont détruites : elles sont toujours satisfaites. Il en est ainsi, par exemple, de : $p(6) \neq p(5) + 1$ ou de $p(6) \neq p(7) + 1$, puisque $p(6) < p(5)$ et $p(6) < p(7)$.

A cette étape, aucune nouvelle information ne peut plus être déduite, mais p est une surjection et le point 9 de l'espace-image n'a plus comme antécédents possibles que $p(5)$ et $p(9)$.

Le choix $p(5) = 9$ est d'abord effectué. Il entraîne immédiatement la suppression de quinze arcs, car autre le fait que $p(i) \neq 9$, pour tout i autre que 5, les contraintes sur a imposent : $p(6) \neq 8$, $p(7) \neq 8$, et les contraintes sur m , avec $p(3) \leq 8$ et $p(7) \leq 7$, imposent :

$$p(2) \leq 7; p(1) \leq 6; p(4) \leq 7; p(6) \leq 6; p(8) \leq 6.$$

Ainsi, seules $p(3)$ et $p(9)$ peuvent valoir 8. **Le choix** $p(3) = 8$ conduit alors à : $p(4) \neq 7$ et $p(7) \neq 7$.

Le point 7 n'a plus que deux antécédents. **Posant** $p(2) = 7$, ALICE en infère : $p(1) = 6$ ou $p(9) = 6$

Posant $p(1) = 6$, il vient alors : $p(4) < 4$ et $p(7) < 4$, donc : $p(6) < 3$ et nécessairement : $p(9) = 5$.

Posant $p(7) = 4$, avec la seule contrainte restante : $p(4) \neq p(6) + 1$, quatre solutions sont atteintes ici :

6 7 8 1 9 2 4 3 5, 6 7 8 1 9 3 4 2 5, 6 7 8 2 9 3 4 1 5, 6 7 8 3 9 1 4 2 5.

Il ne faudra revenir que sur **six choix binaires, soit 64 possibilités** au lieu de 91 = 3.628.80 pour construire les 42 solutions effectives du problème.

Cela dit, il existe des cas pires encore pour le programme spécifique, par rapport à ALICE. Supposons un instant que les données soient, par exemple :

$$n = 100 \text{ et } m \equiv a \equiv (1 \text{ suivi de } 98 \text{ zéros}).$$

La procédure PERM travaille toujours de la même manière. Elle engendre successivement les vecteurs : 1, 12, 123, 13, 132, 1324, 14, 142, 1423, 143, 1432, 14325 et ainsi de suite, alors qu'en réalité, comme le déduit immédiatement ALICE, par analyse des contraintes en \leq , on a nécessairement :

$$p(2) > p(l) \text{ et } p(2) > p(3) > p(4) > \dots > p(100) \text{ soit } \mathbf{p(2) = 100}.$$

De plus, toutes les contraintes sur les avances sont trivialement satisfaites, puisque pour $j \geq 2$ et $i \geq 3$ on a : $p(j) < p(i)$. Donc, le seul degré de liberté concerne la valeur de $p(i)$ dans l'ensemble [1, 99]. Il y a exactement 99 permutations qui conviennent. ALICE les donne ici en un **temps polynômial** avec n .

Bien sûr, ce cas de données peut être spécialement prévu dans PERM. Mais on peut ainsi compliquer à loisir : placer quelques 1 ailleurs dans a (et le problème n'a plus de solution), échanger les 1 en 0 et réciproquement. **PERM. même modifié, sera toujours en retard d'un jeu de données.**

La situation est encore plus grave si l'on ajoute des contraintes nouvelles au problème pour le modifier légèrement, comme cela arrive en réalité bien souvent.

SI l'on impose par exemple : $p(\frac{n}{2}) = 1$, la procédure PERM sera bien difficile à modifier pour prendre en compte assez tôt ce nouveau renseignement, tandis qu'ALICE saura immédiatement l'utiliser et ainsi abréger sa recherche.

8.9 Saisie des énoncés en français

Nous travaillons actuellement à rendre l'entrée des énoncés la plus naturelle possible et à mettre le système à la disposition de tout utilisateur même non informaticien. Il s'agit essentiellement de doter le système de connaissances de sens commun, de manière à éviter, dans le futur, à l'utilisateur d'avoir à préciser les objets courants et leurs propriétés. L'idéal est alors la saisie directe en français avec une lecture interactive et testée à mesure.

C. Lopez-Laseica a ainsi réalisé un prototype qui va maintenant être décrit. La grande difficulté, ici, est l'énorme masse de connaissances « courantes » que nous utilisons dans tous les domaines pour nous exprimer et communiquer. Tout informaticien sait que son travail le plus complexe est avant tout de comprendre l'interlocuteur...

Avec C. Lopez-Laseica, nous avons volontairement limité, pour un premier essai, l'univers du discours et nous avons retenu celui des casse-tête mathématiques. Il n'est sans doute pas représentatif de tous les énoncés possibles, mais il présente une difficulté essentielle : il faut comprendre l'énoncé en profondeur, trouver les liens qui unissent des phrases éventuellement très distantes, afin d'être à même de poser, puis de résoudre le problème. Le langage ALICE sert ici de représentation pivot, suffisamment concise et expressive pour qu'avec les techniques (dictionnaires sémantiques et pragmatiques, réseaux sémantiques) actuellement disponibles en Intelligence Artificielle, la compréhension et la traduction se fassent efficacement : une à quatre secondes pour des textes tenant sur une demi-page.

8.9.1 Principe de la traduction

La partie principale du **programme de traduction est dans les données** : le dictionnaire indique pour chaque mot et pour chaque acceptation d'un mot ses formes syntaxiques, ses classes et attributs sémantiques, les mots auxquels il peut être relié dans l'univers habituel, les inférences de bon sens qu'il peut induire. Ainsi trouve-t-on dans le dictionnaire :

FRANCE	
définition	= élément
groupe	= pays
inférences	= langue : français, nationalité : français, régime : république ;
relation	= habitant, état

Dans ce dictionnaire, qui comprend un millier de mots pour cinquante énoncés traités, les verbes jouent un rôle particulier. Il vont en effet indiquer les actions à entreprendre pour exprimer les contraintes. Ainsi nous avons pour le verbe « donner » :

DONNER	
définition	= verbe
groupe	= personne à personne ;
inférences	= avant : (sujet de donner) possède (chose donnée), aprsès : (sujet de donner) possède (- chose donnée), (objet de donner) possède (+ chose donnée).

Dans l'entrée du mot ECHANGER qui correspond à la construction syntaxique « A échange avec B, X contre Y » on trouvera alors :

ECHANGER (A B) (X Y)	
définition	= A donne X à B et B donne Y à A.

L'acception convenable de tous les mots est déterminée de proche en proche par le programme, qui analyse le texte à l'aide du dictionnaire. La syntaxe et la sémantique participent concurremment à la compréhension.

Les verbes donnent l'ossature principale de la phrase (phase 1). Ensuite, les petits mots, les modificateurs sont traités. Ils donnent leurs articulations aux groupes syntaxiques les uns par rapport aux autres (phase 2). La partie superflue de l'information issue du dictionnaire, mais inutile dans le contexte, c'est-à-dire ne générant aucune contrainte, est alors **oubliée** : l'adjectif « gris » dans « un chat gris » a de l'importance si plus tard on parle d'un « chat roux » ; mais inversement, il n'en a pas, si aucune couleur n'est plus jamais mentionnée.

Dans une troisième phase, le programme engendre effectivement la traduction en ALICE.

Notons que c'est seulement lorsque tout le sens du texte a été correctement interprété que la traduction peut être effectuée. Suivant le contexte, la phrase : « Jean et le professeur lisent » devient :

activité (Jean) = activité (professeur) = lecture

ou bien seulement : Jean ≠ professeur.

Le programme décide, suivant le reste de l'énoncé, le sens utile de la phrase dans le casse-tête. Nous présentons simplement ici brièvement le traitement d'un exemple.

8.9.2 Exemple de traduction

L'énoncé est donné en français à la console :

Enoncé :

Cinq personnes de nationalité différentes habitent les cinq premières maisons d'une rue.

Elles exercent cinq professions différentes et ont chacune une boisson, un animal favori, tous différents. Les cinq habitations sont de cinq couleurs différentes :

*L'anglais habite la maison rouge.
L'espagnol possède un chien.
Le japonais est peintre.
L'italien boit du thé.
Le norvégien habite la première maison à gauche.
Le propriétaire de la maison verte boit du café.
Cette maison verte est à droite de la blanche.
Le sculpteur élève des escargots.
Le diplomate habite la maison jaune.
On boit du lait dans la maison du milieu.
Le norvégien habite à côté de la maison bleue.
Le violoniste boit des jus de fruits.
Le renard est dans la maison voisine du médecin.
Le cheval est à côté de celle du diplomate.*

Qui élève le zèbre et qui boit de l'eau ?

L'interpréteur isole d'abord, dans la première phrase, le groupe nominal : l'ensemble « personne » est créé dans le réseau sémantique avec deux attributs (« 5 » et « nationalités différentes ») ; le verbe « habite » met ce groupe en relation avec celui qui suit ; « les », parce qu'il est suivi d'un adjectif est pris comme article. L'ensemble « maison » est créé avec ses attributs. Toutes les phrases sont successivement représentées et viennent compléter le réseau interne.

Les notations sont alors standardisées et le réseau est simplifié : « habiter », « être dans », « posséder » deviennent « avoir pour maison » ; « à gauche », « à côté », « au milieu » sont exprimés par des relations numériques. Finalement, le sens des mots est tout entier dans le réseau sémantique et il suffit, avant de passer en ALICE, de conserver la correspondance suivante :

NAT(1)	\leftrightarrow	Anglais	BOI(1)	\leftrightarrow	Thé
NAT(2)	\leftrightarrow	Espagnol	BOI(2)	\leftrightarrow	Café
NAT(3)	\leftrightarrow	Japonais	BOI(3)	\leftrightarrow	Lait
NAT(4)	\leftrightarrow	Italien	BOI(4)	\leftrightarrow	Eau
NAT(5)	\leftrightarrow	Norvégien	BOI(5)	\leftrightarrow	Jus de fruit
MAI(1)	\leftrightarrow	Rouge	ANI(1)	\leftrightarrow	Chien
MAI(2)	\leftrightarrow	Verte	ANI(2)	\leftrightarrow	Escargot
MAI(3)	\leftrightarrow	Blanche	ANI(3)	\leftrightarrow	Renard
MAI(4)	\leftrightarrow	Jaune	ANI(4)	\leftrightarrow	Cheval
MAI(5)	\leftrightarrow	Bleue	ANI(5)	\leftrightarrow	Zèbre
PRO(1)	\leftrightarrow	Peintre			
PRO(2)	\leftrightarrow	Sculpteur			
PRO(3)	\leftrightarrow	Diplomate			
PRO(4)	\leftrightarrow	Violoniste			
PRO(5)	\leftrightarrow	Médecin			

Toutes les contraintes sont maintenant exprimées moyennant cette correspondance et l'énoncé final en ALICE est communiqué au système de résolution :

Soit

ENS	ES	INT	=	1	5
	ES	INT	=	1	5
	NA	INT	=	1	5
	MA	INT	=	1	5
	PR	INT	=	1	5
	SM	INT	=	1	5
	AN	INT	=	1	5
	CO	INT	=	1	5

Trouver

BIJ	NAT	NA	\leftarrow	ES
	MAI	MA	\leftarrow	ES
	PRO	PR	\leftarrow	ES
	BOI	BO	\leftarrow	ES
	ANI	AN	\leftarrow	ES

Avec

NAT	1	=	MAI	1
NAT	2	=	ANI	1
NAT	3	=	PRO	1
NAT	4	=	BOI	1
NAT	5	=	1	
MAI	2	=	BOI	2
MAI	2	=	MAI	3 + 1
PRO	3	=	ANI	2
PRO	3	=	MAI	4
BOI	3	=	3	
NAT	1	=	MAI	1
(NAT 5 = MAI 5 + 1) ou (NAT 5 = MAI 5 - 1)				
PRO	4	=	BOI	4
(ANI 3 = PRO 5 + 1) ou (ANI 3 = PRO 5 - 1)				
(ANI 4 = PRO 3 + 1) ou (ANI 4 = PRO 3 - 1)				

La solution de ce casse-tête est donnée en un seul choix après quatre secondes d'exécution.

Certes, la plupart des problèmes réels possèdent des énoncés plus longs, plus touffus et plus ambigus. Nous avons seulement voulu montrer qu'à moyen terme un système complet de résolution de problèmes, conversationnel, partant d'énoncés en français courant était possible.

Chapitre 9

L'apprentissage

L'apprentissage est défini par **l'amélioration des performances avec l'expérience**.

C'est certainement le domaine d'avenir de l'intelligence artificielle : il y a tant de connaissances diverses à posséder, que l'idéal est qu'un système informatique les apprenne de lui même à partir des exemples plutôt que de les recevoir de nous une à une. Cependant, pour le court terme, il est illusoire de vouloir faire un programme d'apprentissage là où nous ne savons même pas comment doit fonctionner un programme sans apprentissage.

Par ailleurs, un homme qui apprend se heurte au problème de l'oubli : beaucoup de possibilités sont engendrées et il faut savoir oublier. Nous touchons là une différence technologique d'importance entre les ordinateurs et nous, vis-à-vis de l'acquisition des connaissances en général, et de la résolution de problèmes en particulier.

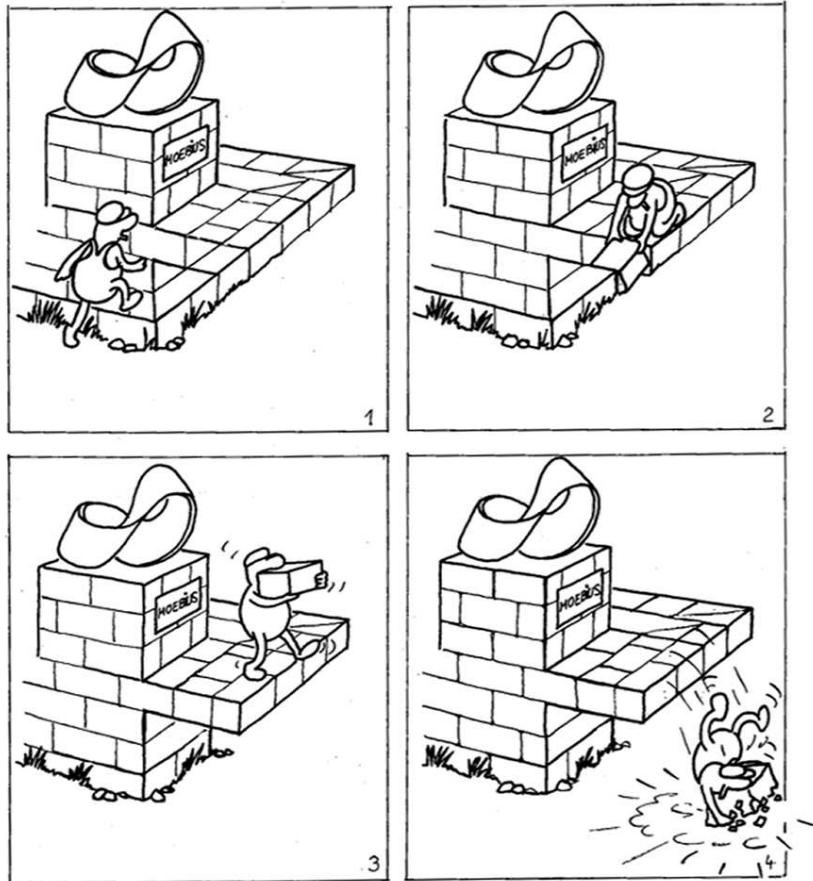
L'homme est préprogrammé pour apprendre.

Il ne sait rien faire sans mémoriser ce qu'il fait ; ceci quelque fois contre sa volonté. Le nouveau né possède déjà des attitudes réflexes et est par nature, fondamentalement curieux. L'homme, toute sa vie, ne peut s'empêcher d'aller à la quête de l'information, de tourner le regard vers quelque chose qui bouge, de tendre l'oreille vers un son inhabituel. L'ordinateur est, a priori, dénué d'un tel programme de recherche et d'apprentissage général . Aussi, les tentatives décrites ci-après sont-elles limitées à des solutions moins spécifiques.

9.1 Différents types d'apprentissages

Soulignons tout d'abord qu'il existe dans l'apprentissage toute une graduation. Le niveau 1 est simplement l'apprentissage programmé : un code commande ce qui doit être fait. Tous les programmes habituels d'ordinateur sont dans ce cas, ainsi que la plupart des robots industriels pour lesquels le bras d'un ouvrier, par exemple, guide une fois pour toutes, les mouvements à exécuter pour peindre une voiture ; le robot les imitera ensuite strictement, qu'il ait d'ailleurs une voiture ou non...

Le niveau 2 correspond à l'apprentissage par cœur : toutes les situations sont mémo-risées avec l'action à entreprendre en face de chacune d'elles. Le système peut être ici conditionné à la Pavlov : il reçoit une punition si, en face d'une situation nouvelle, il ne trouve pas la bonne action et une récompense sinon.



Une amélioration de cette méthode est constituée, au niveau 3, par l'apprentissage statistique : le système ne retient qu'une classification des situations, effectuée à partir d'un grand nombre de cas : seules les composantes pertinentes face à une action déterminée sont mémorisées. Viennent ensuite deux cas intéressants : au niveau 4 le système est capable, à l'aide "d'un professeur" d'apprendre à partir d'exemples. Il sait généraliser.

Enfin au niveau 5, l'apprentissage a lieu sans professeur, le système, par une démarche purement inductive, sait créer de nouvelles situations, de nouvelles hypothèses, de nouveaux concepts.

Notons que, quel que soit le niveau, il ne saurait être question d'apprendre à partir de rien. C'est précisément les connaissances minimales pour apprendre, dans l'univers où nous sommes, qui nous font à présent défaut pour programmer un système vraiment capable d'apprendre.

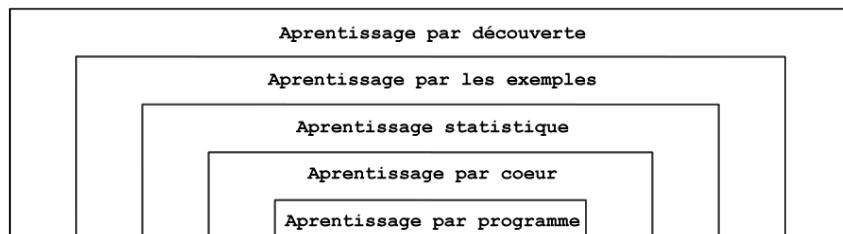


FIG. 9.1 – Différents types d'apprentissage

9.2 Apprentissage de paramètres au jeu de checkers

Les checkers sont les dames anglaises. A. Samuel, ingénieur chez IBM écrivit pour ce jeu de merveilleux programmes entre 1966 et 1967. On joue au checkers sur un échiquier 8×8 . Les pions ne peuvent qu'avancer et les dames ne sautent qu'une case à la fois en avant ou en arrière. Le facteur de branchement vaut alors environ 8.

9.2.1 Apprentissage par cœur

Le premier programme écrit par Samuel est de niveau 2. Samuel a rentré environ 180 000 positions tirées des meilleurs livres pour éduquer ce programme !

Comme cela est beaucoup, surtout pour les ordinateurs de l'époque, Samuel a recours à trois procédures de gestion de la mémoire :

- a) Les situations sont triées en vue d'un accès rapide :
 - Le nombre de pions et de dames, de part et d'autre, les distinguent, ainsi que :
 - La fréquence d'apparition, dans l'expérience propre du programme.
- b) Les situations symétriques en changeant blanc en noir, sont traitées simultanément.
- c) Enfin une procédure d'élimination des positions les moins souvent rencontrées dans les parties les plus récentes, entre en jeu dès que la mémoire est saturée (Samuel inventait ainsi 15 ans avant, un algorithme de remplacement de pages pour les ordinateurs à mémoire virtuelle).

En outre, et d'aucuns prétendent, que ceci n'est pas étranger à la taille des mots sur les IBM des séries 700 et 7000, Samuel avait su trouver une bonne représentation du damier : les pions ne jouent que sur les cases noires . Samuel numérote ainsi l'échiquier 8×8 élargi :

	1		2		3		4
5		6		7		8	
9	10		11		12		13
	14		15		16		17
18		19		20		21	
	23		24		25		26
27		28		29		30	
	32		33		34		35
							36

FIG. 9.2 – Représentation du damier

Les quatre cases fantômes 9, 18, 27 et 36 servent de garde fous : il suffit de représenter l'échiquier comme un vecteur $[1, 36]$ sur un mot pour trouver les coups légaux (sans prise) des noirs par addition de 4 et de 5 (décalage puis masquage).

Mais, ce premier programme ne fut que le point de départ pour un second, qui l'améliorait . Il était en effet dommage de perdre toute cette information dès que la situation exacte n'était pas en mémoire. Samuel introduisit donc un arbre de recherche pour étudier la position la plus avant. Celui-ci fut limité à trois demi coups sauf en cas de

prise au dernier coup ou si un pion noir est en prise (ces restrictions poussent quelquefois la recherche à vingt demi coups). Enfin Samuel utilisa une fonction d'évaluation F polynômiale :

$$F = \sum p_i P_i$$

Il choisit 38 paramètres p_i et 38 caractéristiques P_i parmi lesquelles : l'avantage en matériel (P_0), le contrôle du centre, le nombre de pions exposés, la mobilité, le nombre de formations en V (vide bordé de 3 pions noirs), le nombre de menaces, l'avance relative des deux camps.

Samuel est ainsi sans doute le premier à avoir programmé la procédure alpha-bêta (cf. chapitre 4.3) qui fit par la suite les beaux jours des programmes d'échecs. Ce programme va maintenant être utilisé de trois manières différentes.

9.2.2 Apprentissage livresque

Fort de toutes les positions misées en mémoire, le programme évalue lors de toute partie, toutes les positions par minimax avec la fonction d'évaluation F . Ce faisant, il comptabilise le nombre H de coups notés meilleurs selon F que le coup mémorisé, et le nombre L de coups notés moins bons. Si H est faible et L élevé. Il y a accord entre le livre et l'évaluation. Dans le cas contraire, le coefficient de « corrélation » C :

$$C = \frac{L - H}{L + H}$$

qui avait la valeur 1 en cas d'accord parfait, peut aller jusqu'à prendre la valeur -1 si le désaccord est total entre le livre et F .

Dans le cas où C n'est pas égal à 1, l'évaluation de la situation a été surestimée si $H > 0$, et sous-estimée sinon. L'apport C_i de chaque paramètre P_i est calculé :

$$C_i = \frac{L_i - H_i}{L_i + H_i}$$

avec : H_i = nombre de coups placés par $p_i P_i$ avant le meilleur avec $p_i < 0$;

L_i = nombre de coups placés par $p_i P_i$ après le meilleur avec $p_i < 0$;

Si C_i est négatif, il faut diminuer les p_i positifs ;

Si C_i est positif, il faut augmenter les p_i négatifs.

Le programme d'apprentissage livresque décide alors de changer le paramètre p_k dont le coefficient c_k a la plus grande valeur absolue. Il pose alors :

$$P_k = 1$$

et pour tout i , $i \neq k$, $p_i = 2^n$ si $n \leq \| \frac{c_k}{c_i} \| \leq n + 1$

Enfin le programme ne conserve que 16 des 38 caractéristiques de départ. A tout moment, $22p_i$ valent 0. A chaque partie le plus petit c_i donne un mauvais point à sa caractéristique : au bout de 3 mauvais points, une caractéristique est chassée et remplacée par celle qui est depuis le plus longtemps sur la liste d'attente. Après une

vingtaine de parties, ce programme livresque était déjà d'un bon niveau. Samuel voulut aller plus loin.

9.2.3 Apprentissage sans professeur

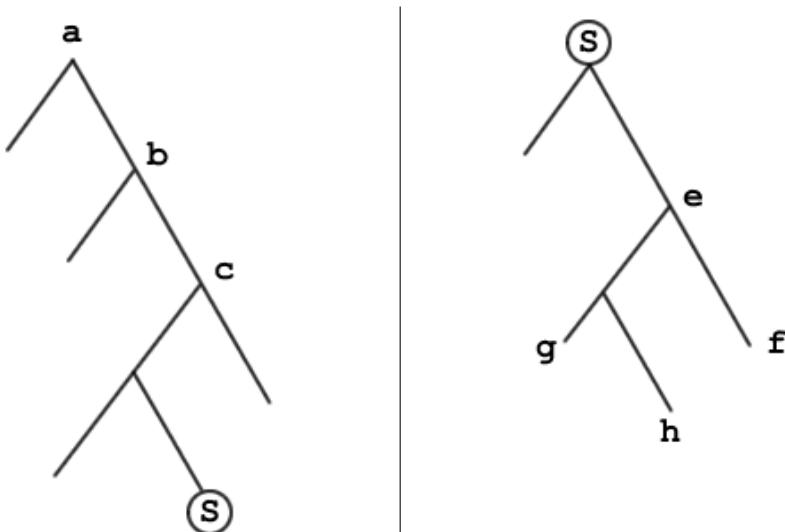
Puisqu'il s'agit d'un jeu, le programme peut jouer contre lui-même. Des valeurs différentes des p_i peuvent alors lui permettre d'apprendre tout seul! Soient A et B , deux versions du programme avec deux fonctions d'évaluation F_a et F_b . Si A gagne, on change un peu les paramètres de F_b . Si A perd, il prend un mauvais point . Après trois mauvais points, on échange F_a et F_b et on change radicalement les paramètres de F_b . C'est une technique classique de gradient avec perturbation pour tenter d'explorer un espace non convexe .

Améliorant cette idée en la combinant avec le *minmax*, Samuel écrivit un quatrième programme .

9.2.4 Apprentissage généralisé

C'est Samuel lui-même qui qualifie ainsi les capacités de ce nouveau programme. Samuel trouvait choquant qu'aucun de ses précédents programme ne puisse **apprendre en cours même de partie**.

Au contraire, nous nous rendons nous-mêmes compte qu'une erreur a été commise, seulement après quelques coups, et évitons qu'elle se reproduise dans la partie en cours. Or, un programme qui fonctionne avec une fonction d'évaluation et un alpha-bêta, possède lui aussi ce jugement a posteriori de tous les coups : il rencontre une situation donnée d'abord en cours d'évaluation et il la retrouve éventuellement quelques coups plus tard, sur l'échiquier réel.



Situation rencontrée lors de la recherche alpha-bêta. Son évaluation est alors $F(S)$.

Même situation rencontrée maintenant effectivement en cours de jeu : évaluation = $F'(S)$.

FIG. 9.3 – Deux évaluations de la même position

Dans le cas idéal, si F est une évaluation parfaite et si tout se passe comme prévu, on devrait trouver :

$$F(S) = F'(S)$$

Si l'on rencontre au contraire $F(S) > F'(S)$, c'est que l'évaluation avait été optimiste et que l'adversaire a été meilleur que ce qui avait été escompté : comme dans l'apprentissage livresque, le programme, dans ce cas, diminue les poids positifs et augmente les poids négatifs. Il suit ici une procédure en tous points semblable à la précédente, calculant à nouveau les coefficients :

$$C_i = \frac{L_i - H_i}{L_i + H_i}$$

où H_i est maintenant, le nombre de fois, depuis le début de la partie, où pour la i ème caractéristique, on a eu :

$$p_i P_i < 0 \text{ et } F(S) > F'(S)$$

Le cas $F(S) < F'(S)$ est, comme ci-dessus, traité symétriquement. Remarquons qu'ici encore « l'adversaire » peut très bien être le programme lui-même. En milieu de partie où les positions sont considérablement plus nombreuses, c'est cette possibilité qui a été utilisée.

Les performances de ce dernier programme sont relativement instables au début, mais Samuel dispose des 180 000 coups déjà rentrées pour le programme précédent ; et, la nuit, le programme joue en outre contre lui-même. Le résultat est que Samuel est maintenant régulièrement battu aux dames par son propre programme.

Un peu plus tard, ce programme, qui atteint ainsi un niveau sans précédent dans les programmes de jeu, gagne une partie contre le champion de l'état du Connecticut (K. Nealey, 1962). Il perd honorablement quatre parties par correspondance contre le champion du monde et obtient une nulle devant le damier (W. Hellman, 1965).

Le programme joue de mieux en mieux et de plus en plus vite (la probabilité augmente pour que la position courante soit déjà stockée en mémoire avec son évaluation). Cependant cet apprentissage généralisé comportait un risque et Samuel a eu de la chance : l'apprentissage dépend de la qualité du jeu de l'adversaire et si celui-ci joue mal, le programme apprendra de mauvais paramètres. En effet, la procédure de *minmax* suppose que le camp ennemi joue au mieux.

Si l'adversaire joue mal, le programme en déduit - indûment - que la position a été sous évaluée et change les paramètres en conséquence.

Comme l'apprentissage a lieu à chaque coup, en cours même de partie, l'ensemble des paramètres peut être complètement perturbé après quinze coups. Le programme lui-même commence alors à mal jouer et il suffit que l'adversaire se mette brusquement à bien jouer pour gagner la partie.

Samuel, le premier apporte des modifications techniques à la procédure alpha-bêta : élagage progressif de l'arbre en largeur et en profondeur, tri des coups au premier niveau. Mais il réfléchit surtout au problème le plus important :

- Par construction, la fonction d'évaluation est linéaire en chacune des caractéristiques. Or, certaines caractéristiques doivent naturellement se renforcer : si P_i et P_j donnent l'avantage aux noirs, un terme en $P_i \times P_j$ doit traduire cet avantage renforcé.
- Par définition, la fonction la plus générale est une table. Il y a ici 38 caractéristiques recensées. Une table, envoyant \mathbb{N}^{38} dans \mathbb{N} serait donc l'idéal.
- Par nécessité, un compromis doit être trouvé entre l'addition linéaire trop grossière et cette table, de taille gigantesque.

Samuel a alors l'idée des *tables de signatures*, tables hiérarchisées qui sont une bonne approximation de la fonction la plus générale :

Les 38 caractéristiques sont regroupées par type où certaines d'entre-elles apparaissent plusieurs fois : dans un même type figurent les caractéristiques jugées corrélées par les experts et par Samuel. Chacune d'elles est limitée à prendre au plus sept valeurs différentes.

En revanche, c'est maintenant pour toute combinaison de valeurs à l'intérieur d'un type que va être appris un coefficient. La figure 9.4 donne l'agencement général de ces tables de signatures.

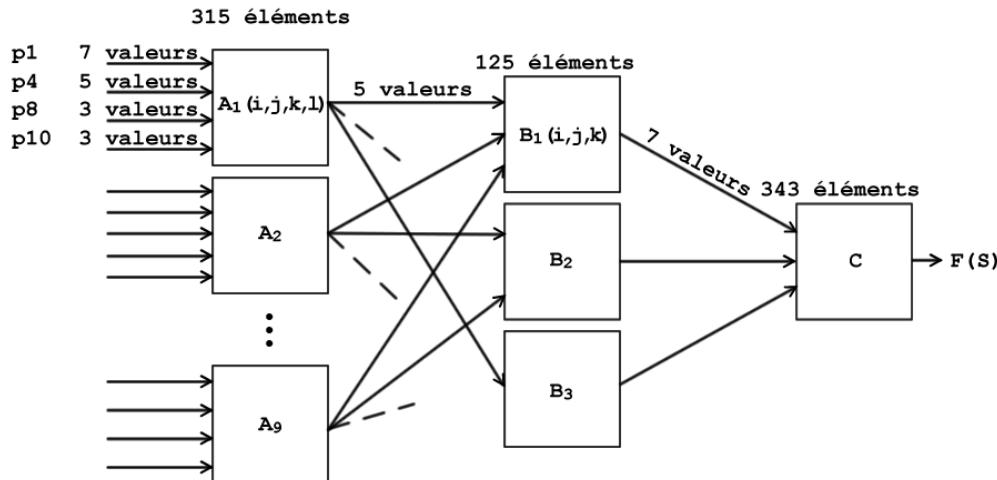


FIG. 9.4 – Hiérarchie de signatures à trois niveaux

Samuel rentre à nouveau ses 180 000 coups d'expert et calcule ses paramètres grâce à la même méthode de corrélation. Il constate qu'après quelques 80 000 coups, tout est déjà stabilisé.

Tout se passe en réalité comme si chaque élément matriciel était un coefficient à lui seul ; il y en a 315×9 pour le type A, 125×3 pour le type B, et 343 pour C soit 3 500 environ au lieu de 16 dans le programme 1.

Comme la plupart des caractéristiques n'interviennent qu'en certaines phases du jeu, toute partie est divisée en six phases, en fonction du nombre de pions encore en piste et de leur avancée. Ce découpage réduit encore le temps de calcul. Notons que celui-ci est très faible puisque l'évaluation n'est qu'une simple recherche en table.

L'évaluation $F(S)$ est donnée directement par une instruction unique :

$$F(S) \leftarrow C(B_1(A_1(p_1, p_2, p_3, p_4), A_2(\cdot), A_9(\cdot)), B_2(\cdot), B_3(\cdot))$$

Ce programme devient et restera un vrai champion dans ce jeu. Sans aucun alpha-bêta.

Il classe déjà dans 68 % des cas le meilleur coup dans les deux premiers (contre 32 % avec une évaluation polynômiale et 16 % en tirant au hasard) et l'alpha-bêta améliore notablement ce score. En outre il permet au programme un jeu régulier - sans mauvais coup - tout au long de la partie.

Si le gros défaut des programmes est de partir à chaque fois de caractéristiques p_i données sans pouvoir les découvrir par eux-mêmes, cet apprentissage de type statistique est bon et est même souvent utilisé par les humains. Une équipe de chercheurs va aller un peu plus loin en écrivant un système capable de construire des ensembles d'actions jugées intéressantes dans une famille de situations. Cette recherche a été développée dans le monde des robots et le système a pour nom STRIPS (Stanford Research Institute Planning System ; Fikes, Nilson 1973).

9.3 Apprentissage d'actions utiles par un robot : STRIPS

Les opérateurs élémentaires sur l'univers sont décrits initialement au robot comme des triplets de listes :

- La première liste donne les conditions dans lesquelles chaque opérateur peut éventuellement être déclenché.
- La seconde et la troisième précisent les faits qui doivent être respectivement retranchés et ajoutés à la description de l'univers si le robot utilise effectivement l'opérateur en question. Les modifications induites sont propagées dans STRIPS à l'aide d'un démonstrateur de théorèmes basé sur la Résolution (voir chapitre 3)

Opérateur 27 : déplacer un objet dans une pièce)
Liste C (conditions) : Robot au sol. Robot dans une pièce x en z_1
Objet O au sol en z_1 , objet O déplaçable.
Places z_1 et z_2 dans pièce x .
Liste - Robot en z_1 . Objet O en z_1 .
Liste + Robot en z_2 . Objet O en z_2 .

FIG. 9.5 – Exemple de description d'opérateur élémentaire dans STRIPS

L'apprentissage a lieu chaque fois que le système a effectivement résolu un problème. Il revient alors sur la suite des opérateurs élémentaires qu'il a utilisé pour tenter d'en déduire une information ajoutée. Un plan qui a réussi en n pas est, en effet une mine potentielle pour des buts intermédiaires, mine que STRIPS va maintenant exploiter. La suite O_1, \dots, O_n de ces opérateurs est d'abord organisée en une table triangulaire (figure 9.6).

On trouve, par construction, dans cette table, à l'intersection d'une ligne i et d'une colonne k les faits ajoutés par l'opérateur k qui ont été conservés jusqu'à l'application de l'opérateur i . Si l'on rencontre alors en ligne j , une description qui coïncide avec celle du but d'un nouveau problème, il suffit d'extraire de la séquence O_1, \dots, O_n la sous séquence utile : la moitié du travail est fait, O_j est le dernier opérateur à utiliser. En remontant la table, il suffit de déterminer le premier O_i dont les conditions sont satisfaites dans le nouvel univers.

La séquence O_i, O_{i+1}, \dots, O_j peut alors servir directement pour atteindre le nouveau but : c'est une macro action , correspondant par exemple à « prendre l'objet O , ouvrir

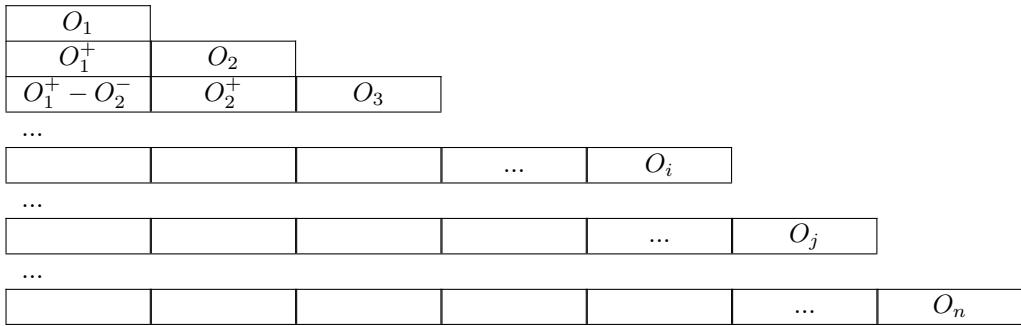


FIG. 9.6 – Séquences d'opérateurs dans STRIPS

une porte, allumer une lumière, monter l'escalier , déposer O , fermer la lumière, fin », que le robot a ainsi apprise et mémorisée.

Un autre programme développé par D. Waterman , à nouveau à propos d'un jeu - le poker - apprend en analysant a posteriori sa propre expérience.

9.4 Apprentissage de règles

Waterman étudie le jeu de poker à deux joueurs lors des enchères sur le pot. Un joueur peut soit relancer en augmentant la mise, soit demander à voir la main adverse en enhérissant à égalité, soit abandonner. Si la main de l'adversaire est vue, la main la plus haute emporte le pot. et il est possible de voir si l'on aurait éventuellement pu gagner plus ou perdre moins. C'est cette information que va utiliser le programme pour modifier ultérieurement ses décisions. La phase d'échange des cartes est résolue ici par un algorithme de façon à augmenter au maximum l'espérance de la valeur individuelle de chaque main, indépendamment de celle de l'adversaire.

L'idée générale est de décrire le jeu par un **vecteur de caractéristiques** (on se souvient de Samuel ...) et les décisions par un jeu de règles de production séparées du programme et modifiables à volonté. Le poker est un jeu à information incomplète qui demande une évaluation tant de la main que du style de jeu de l'adversaire. Les situations sont moins bien définies qu'aux checkers et les règles de production vont montrer toute leur utilité par rapport aux tables de signatures. La procédure *minmax* ne peut plus en effet ici jouer son rôle habituel de garde-fou : chaque coup est unique et ne sera jamais plus rencontré dans la partie. Le *vecteur descriptif* du jeu comporte essentiellement six composantes :

- la valeur de la main (VM)
- la hauteur du pot (HP)
- la dernière relance adverse (DR)
- le nombre de cartes échangées par l'adversaire (CE)
- la probabilité de pouvoir bluffer l'adversaire (PB)
- le style de celui-ci : plus ou moins conservatif (CA)

En outre, des composantes statistiques sur l'ensemble des parties avec cet adversaire sont calculées par le programme : corrélations entre ses enchères et la hauteur effective des mains, nombre de bluffs constatés, nombre d'abandons. Les règles de production sont de la forme :

vecteur caractéristique de la situation → enchère à effectuer

Ces règles sont ordonnées. La règle i ne se déclenche que si les règles 1 à $i - 1$ n'ont pu être déclenchées.

Une règle comme la suivante, donnée par un professeur :

« *Un joueur qui détient une main sûrement gagnante doit faire l'enchère la plus haute possible sans que l'adversaire risque d'abandonner. Cependant, si le pot est déjà très haut alors il faut demander à voir.* »

sera traduite pour le programme sous la forme (une étoile signifie que le paramètre associé n'a pas d'importance) :

$$R_1 : (SG^t, *, r, *, PB, CA) \rightarrow \text{Max}$$

$$R_2 : (SG^t, T_{haut}, r, *, *, *) \rightarrow \text{VOIR}$$

R_2 sera examinée **avant** R_1

La valeur T_{haut} est fixée par ailleurs, par exemple, $T_{haut} = 100$. La présence de r signifie que l'adversaire a effectivement relancé ($r > 0$).

La valeur de Max , enchère maximale sans risquer l'abandon adverse, est fixée par une autre règle et dépend notamment des valeurs de PB et CA . Enfin, une telle règle peut être affinée ou modifiée par le programme. L'apprentissage, aidé dans un premier temps par un professeur, va procéder par renforcement : les bonnes règles sont favorisées, les autres voient restreindre leurs applications.

L'information disponible après tout coup est supposée donnée **par un professeur** :

- vecteur courant des variables pertinentes ;
- bonne décision ;
- raison de cette décision.

Trois cas peuvent alors se présenter :

- a) la bonne décision a été prise par le programme et pour les bonnes raisons, rien n'est modifié dans ce cas ;
- b) aucun membre droit ne correspond à la bonne décision et la i ème règle avait été déclenchée : le programme intercale en ($i - 1$) la règle indiquée par le professeur ;
- c) il existe avant ou après la règle déclenchée, R_d , une règle R_j , qui pouvait donner la bonne décision : les paramètres des règles qui ont empêché le déclenchement de R_j sont automatiquement modifiés.

Exemple :

Les règles sont actuellement :

$R_1(a_1, *, c_2, *, *, *)$	→ relancer de 5
$R_2(a_1, b_1, *, *, e_1, *)$	→ augmenter (PB) de 0.5 (probabilité de bluffer l'adversaire)
$R_3(a_2, *, c_1, *, e_1, *)$	→ relancer de 5
$R_4(a_1, *, *, *, *, *)$	→ augmenter (PB) de 0.5

Avec des contraintes de définition imposant : $a_1 > 8$, $a_2 < 10$, $b_1 > 9$, $c_1 < 3$, $c_2 > 11$ et $e_1 = 0$ tandis qu'à la situation est associé le vecteur : $(9, 4, 5, 7, 0, 5)$ La valeur de (VM) correspond donc à un a_1 ou un a_2 ; (HP) et (OR) ne sont ni des b_1 , ni des c_1 ou c_2 et seule la règle R_4 peut s'appliquer. Mais, le professeur critique ici cette décision et dit qu'il fallait louer une relance de 5 car (VM) était grand (supérieur à 7) alors que (OR) était petit (inférieur à 6).

La règle apprise est donc : $R_p(a_3, *, c_3, *, *, *) \rightarrow$ relancer de 5 ; avec $a_3 > 7$ et $c_3 < 6$.

Le programme cherche, dans l'ordre, une règle qui aurait pu conduire à cette action. R_1 est candidate mais ne peut convenir car les définitions de c_2 et de c_3 vont en sens contraire et ne peuvent être ajustées. La règle R_3 fait par contre figure de bonne candidate : la décision était la même et les plages de valeurs peuvent s'accorder : comme cette règle cible R_3 est placée avant la règle fautive R_4 , il faut modifier le champ des paramètres a_2 et c_1 de façon à ce qu'ils puissent correspondre avec la situation.

La règle R_3 est changée automatiquement en : $R'_3(a_4, *, c_3, *, e_1, *) \rightarrow$ relancer de 5 avec $7 < a_4 < 10$ et $c_3 < 6$.

La procédure complète d'apprentissage est donnée par la figure 9.7 :

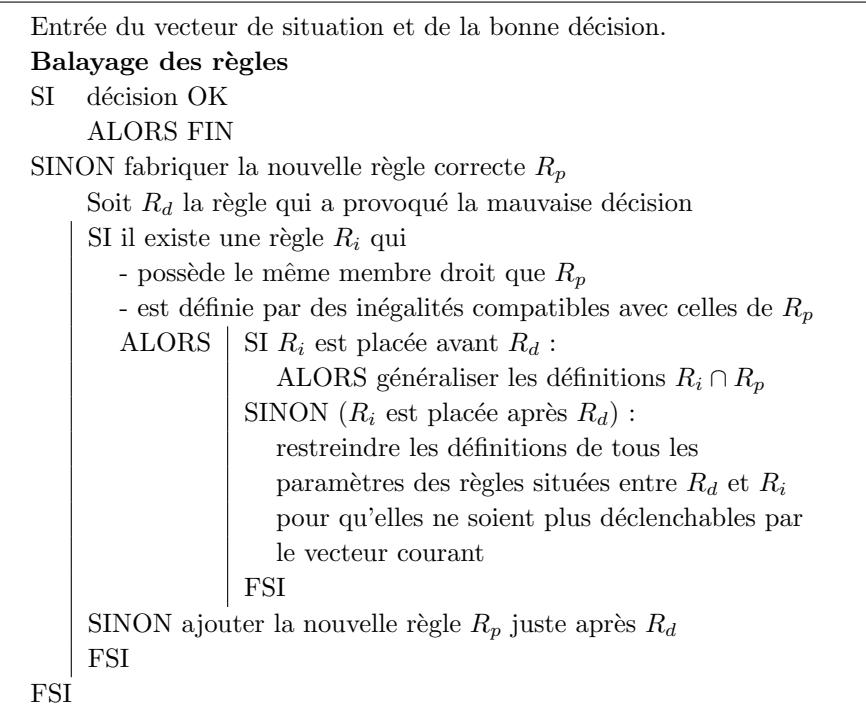


FIG. 9.7 – Procédure d'apprentissage des règles

Deux dangers opposés s'infiltrent dans ce beau processus :

Le premier est une sur-restriction des règles. Ceci se produit avec un professeur trop pointilleux. Le nombre de règles croît indéfiniment, chacune d'elles s'appliquant à un seul cas. Pour éviter cela, le nombre de règles est arbitrairement limité et la dernière d'entre elles, avec le vecteur $(*, *, *, *, *, *)$, donne une décision par tirage au sort. C'est d'ailleurs par cette règle unique que débute ici tout apprentissage. Le second danger est la redondance : une règle R_1 peut rendre toute règle R_2 , qui la suit. Inutile parce que les ensembles de variation des paramètres pertinents de R_2 sont inclus dans ceux de R_1 et ceci quelles que soient les décisions de R_1 et R_2 , ou bien, inversement, R_1 rend inutile une règle R_0 qui la précède car la même décision serait prise par le système si R_0 n'existe pas.

Cette redondance est périodiquement examinée et éliminée par le système (Ce processus est enclenché non pas après tout coup, mais après chaque partie, car les règles peuvent être modifiées rapidement).

Exemple :

$$\begin{array}{ll}
 R_1(a_1, b_1, *, *, *, *) & \rightarrow d_1 \\
 R_2(a_2, b_2, c_1, *, *, f_1) & \rightarrow d_2 \\
 R_3(*, b_2, *, *, f_2) & \rightarrow d_3 \\
 R_4(*, b_1, *, *, *, *) & \rightarrow d_3
 \end{array}$$

La règle R_1 rend R_2 (sous-) redondante car $a_2 > a_1$ et $b_2 < b_1$. R_2 est supprimée. La règle R_4 rend R_3 (sur-) redondante : $b_2 < b_1$ et f_2 n'intervient pas : R_3 est supprimée.

Remarquons que tout est maintenant tout aussi bien en place pour un apprentissage **sans professeur** : à chaque fois que la main de l'adversaire est vue, si le programme perd le coup, il fabrique lui-même une règle qui transforme la dernière enchère en abandon : si au contraire, il gagne, il élève la dernière relance.

Waterman conçut ainsi en fin de compte cinq programmes différents de poker, ils jouent :

- a) au hasard,
- b) en apprenant avec professeur,
- c) par apprentissage autonome,
- d) avec des règles données par un bon joueur,
- e) par apprentissage total sans la donnée des caractéristiques du vecteur de situation.

Le nombre de règles ne dépasse jamais trente et l'accord avec les experts atteint -après apprentissage - 96 % (programme autonome). Le programme change assez souvent de décision - par construction - pour des situations très semblables et bliffe facilement - tout cela sans broncher- ; quelques adversaires sont déconcertés. Les programmes sont testés par lot de 25 mains contre des hommes et un peu plus tard les jeux sont inversés.

Les programmes gagnent facilement contre des joueurs moyens et se comportent presque à égalité avec les experts.

Remarques :

Le travail de Waterman comporte deux points faibles. Tout d'abord, comme chez Samuel, seuls sont finalement appris des paramètres numériques, les éléments symboliques déterminants sont donnés a priori.

En second lieu, l'ordre figé des règles complique le travail du programme et provoque des phénomènes bizarre d'instabilité : ce jeu de règles ordonné est en fait semblable à une procédure et toute modification provoque des effets de bords incontrôlables. L'avantage des règles de production est entièrement perdu (cf chapitre 7). La problématique essentielle de l'apprentissage se dégage maintenant clairement : **ce sont les concepts qu'il faut d'abord construire**. Il faut être capable d'insérer, éventuellement à partir d'un cas unique, le caractère déterminant de la situation. C'est bien ainsi en effet que l'homme procède et n'a besoin de lui expliquer deux fois de ne pas mettre une main dans le feu. Un programme concernant à nouveau un jeu -le jeu d'échecs- va s'attaquer à cet apprentissage à partir d'un seul exemple en le comprenant et en le généralisant.

9.5 Apprentissage de plans

Ce programme de Jacques Pitrat reçoit des parties réellement jouées par des maîtres, il s'efforce de les comprendre et en tire alors **des plans de jeu** réutilisables pour d'autres parties. Au départ, il ne connaît que deux stratégies élémentaires :

- si une pièce est attaquée par l'ennemi, la déplacer ;
- si une pièce amie P_1 attaque une pièce ennemie P_2 , considérer le coup $P_1 \times P_2$

Fort de ces stratégies élémentaires, le programme va essayer de comprendre les attaques et les défenses de chaque camp et il y parviendra, même lorsque les menaces ne passent pas à exécution, en mémorisant les suites de coups qui conduisent à des bénéfices si les ripostes ne sont pas pertinentes . La démarche du programme se déroule en trois temps.

- compréhension de la situation ;
- simplification et généralisation de la séquence ;
- mémorisation et utilisation du plan trouvé.

Compréhension de la situation :

Deux cas intéressants peuvent se présenter :

1. Le dernier coup joué est la capture d'une pièce ennemie. Considérons par exemple, la figure 9.8 et les trois coups suivants qui sont :

- I. $Fb5 \times c6$
- II. $d7 \times c6$
- III. $Cf3 \times e5$ (coup à comprendre)

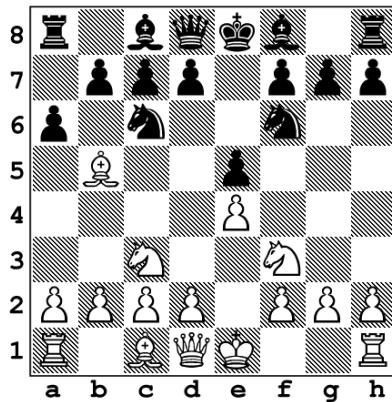


FIG. 9.8 – Un début de partie, trait aux blancs

En réalité , pour comprendre le coup III, il faut comprendre le coup I qui l'a préparé, ainsi que II qui l'a contré. C'est pour ces raisons, qu'afin de comprendre un tel coup de capture, le programme engendre dans un tel cas de façon systématique l'arbre de la figure 9.9.

Dans le début de partie qui nous occupe, nous trouvons en IV (par *minmax*) les coups noirs : $C \times Ce5$ ou bien $a6 \times Fb5$; en V nous rencontrons : $b \times c6$ (sinon les blancs ont

gagné une pièce) et en VI, les noirs se rendent compte qu'ils ont perdu un pion. La leçon de la situation (nous verrons un peu plus loin comment le programme la met à jour puis la retient) est donc : « pour capturer un pion protégé par une seule pièce, échanger d'abord cette pièce ».

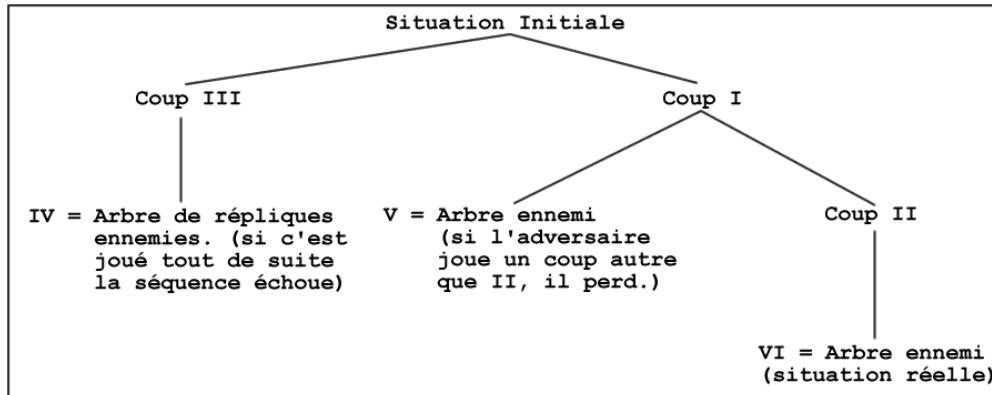


FIG. 9.9 – Arbre de compréhension d'un coup aux échecs

2. Le second type de coup auquel le programme s'intéresse est constitué des *menaces*. c'est à dire, des mouvements qui gagnent une pièce au coup suivant si l'ennemi n'agit pas. Le programme constate la menace, voit comment et pourquoi le camp adverse pare celle-ci et apprend, même si rien de tangible ne s'est passé réellement sur l'échiquier .

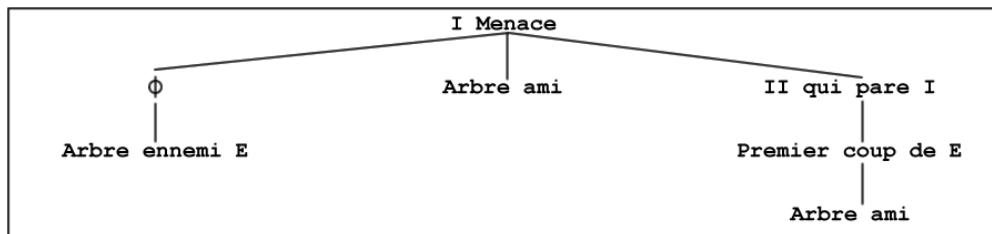


FIG. 9.10 – Arbre d'étude d'une menace

Le programme fabrique et mémorise donc des actions potentielles, apprenant à la fois à attaquer et à défendre, même si sur l'échiquier réel aucune combinaison n'était possible.

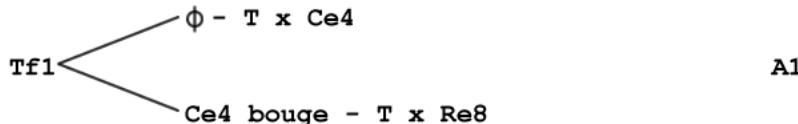
Dans chaque cas ci-dessus, la procédure du *minmax* explique les coups joués dans la mesure où elle donne pour tous les autres coups un bilan négatif .

Simplification et généralisation.

C'est l'arbre de recherche tel qu'il a été présenté ci-dessus et tel qu'il a été précisé par le jeu courant qui est retenu. Mais, il y a peu de chances de rencontrer deux fois la même situation aux échecs : il ne faut retenir que les éléments caractéristiques. Ceci est plus difficile qu'aux checkers ou qu'au poker, car le moindre pion change tout ; Il n'est pas question de jouer de la même façon dans la figure 9.8 si e5 est en e6. Une procédure de simplification ne garde donc tout d'abord dans l'arbre que les meilleurs coups de chaque côté. Seuls les coups qui créent ou détruisent des possibilités de capture sont en fait ici conservés. Dans un second temps, le programme généralise l'arbre de coups qu'il vient de construire avec son bilan positif pour l'un des deux camps. En premier lieu, la couleur du camp n'intervient pas et devient un paramètre à la racine de l'arbre. En second lieu, les cases et les pièces elles-mêmes deviennent des variables :

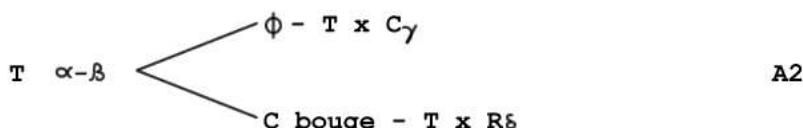
- le nom des cases n'a pas d'importance, seules les relations entre cases et les conditions de positions de pièces sont mémorisées. Ainsi l'échiquier initial :

Blancs : $Rg1$, $Tf1$, $d2$; Noirs : $Re8$, $Ce4$, conduit d'abord à l'arbre A_1 :



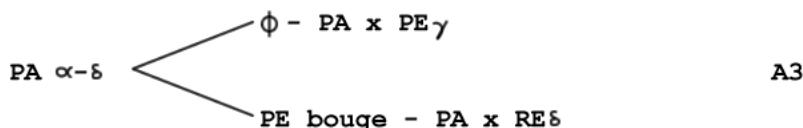
Cet arbre est généralisé en posant :

Conditions initiales : T amie en ∞ , Cavalier ennemi en γ , Roi ennemi en δ ; β, γ, δ , sur la même rangée, avec, maintenant, le nouvel arbre A_2



qui traduit le clouage d'un cavalier par une tour.

- le nom des pièces peut lui-même, sous certaines conditions, être généralisé : toute pièce prise à l'ennemi peut d'abord être changée en une pièce de valeur égale ou plus forte, en conservant un bilan favorable. Inversement, toute pièce amie peut être changée en une pièce amie plus faible. L'arbre A_2 est ainsi généralisé pour donner A_3 :



PA tient lieu d'une pièce amie de valeur inférieure ou égale à une tour ; PE est une pièce ennemie, de valeur supérieure ou égale à un cavalier ; RE est le roi ennemi.

Le concept de clouage est ainsi retrouvé par le programme à partir d'un seul cas.

Dans certains cas, quand une pièce est, par exemple simplement capturée sans intervenir dans le jeu, la généralisation va plus loin et donne à cette pièce une valeur absolument quelconque, éventuellement nulle : sa case est alors considérée comme inoccupée .

Les arbres, ainsi généralisés sont de véritables plans, chacun d'entre eux correspond à un gain matériel, à des suites d'attaques et de défenses ; il peut être utilisé indifféremment par chaque camp. Mais apprendre ne suffit pas, il faut savoir représenter les connaissances apprises pour pouvoir les utiliser efficacement : plusieurs milliers de plans différents vont être engendrés ; leur structuration -et peut-être à terme l'oubli d'une partie d'entre-eux - est indispensable .

Modification et utilisation des plans.

Les arbres généralisés vont être mis en mémoire sous une forme analogue aux schémas de M. Minsky (1972) : chaque schéma est composé de trois parties distinctes, les coups généralisés, les cases, l'arbre lui-même . Une case est repérée par un nom formel considéré comme variable, et par un état : $V = vide$, $A = ami$, $E = ennemi$, avec indication de la valeur dans les deux derniers cas. Les coups généralisés sont définis

par la valeur (généralisée) et le coût de la pièce, les cases de départ et d'arrivée et éventuellement des conditions sur des cases intermédiaires .

Exemple :

$TAa - d - \times E \geq F$ signifie qu'une tour amie en A prend en e une pièce ennemie valant au moins un fou et qu'il existe au moins une case intermédiaire vide - d . L'arbre est enfin mémorisé. Il porte, en chaque sommet, le numéro du coup associé. En face d'un échiquier donné, il faut maintenant trouver tous les schémas applicables. Mais, cela peut coûter très cher si l'on n'y prend garde : Supposons que le premier coup marqué dans l'arbre pour un certain schéma soit : $TA\alpha - \beta$.

Alors, si l'échiquier courant comprend une tour , la variable α peut recevoir une valeur après quoi β peut aller jusqu'à en recevoir quatorze. Il convient donc d'organiser la recherche de façon à éliminer le plus vite possible les plans indéclenchables et ce en faisant le minimum d'essais. Dans un premier temps . seuls sont conservés les plans qui correspondent potentiellement au matériel présent.

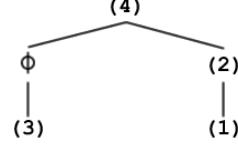
L'idée est ensuite de considérer d'abord les coups généralisés les moins probables, c'est à dire les plus informants. Le coup le moins probable est déterminé dans chaque schéma par une petite fonction d'évaluation de la forme :

$$\min(\text{valeurs cases du coup}) \times [\text{valeur pice}, \min(\text{valeur autres cases})]$$

La valeur d'une case est d'autant plus grande que la case est moins connue ; elle vaut : 32 pour une case vide, 8 pour une case occupée par un pion, 4 pour une case occupée par une pièce, 1 pour la case du Roi, de la Dame ou pour une case déjà connue.

Le coup le moins probable est déterminé par le minimum de cette fonction et, dans le schéma lui-même, les coups sont ordonnés suivant cette fonction.

Ainsi, le plan de l'arbre A_2 conduit au schéma utile :

CASES		COUPS ORDONNÉS		ARBRE
α	TA	(1) TA	$\beta - \gamma \times \delta$	
β	vide	(2) $E \geq C$	joue	
γ	$E > C$	(3) TA	$\beta \times \gamma$	
γ	RE	(4) TA	$\alpha - \beta$	 <pre> graph TD A((4)) --> B((1)) A --> C((2)) A --> D((3)) C --> E((3)) </pre>

L'échiquier examiné est celui de la figure 9.11 où le trait est aux noirs.

L'ensemble « Cases » de ce schéma indique tout d'abord que tout le matériel est présent ; il impose ensuite la valeur de la variable 6 soit $h2$. L'ensemble « Coups », s'il était ordonné suivant l'arbre réel, commencerait par (4) $T\alpha - \beta$ et conduirait à quatorze essais. Au contraire, ordonné grâce à la fonction d'évaluation précédente. Il demande d'abord l'examen de : (1) $TA\beta - \gamma - \times h2$.

L'examen de la position, par analyse des coups de tour qui viennent en $h2$ en interceptant la case initialement occupée par une pièce ennemie au moins égale à un cavalier, retient seulement (1) $TAa2 - c2 - \times h2$ puisque dans $Th8 - h4 - h2$ le pion blanc $h4$ n'a pas une valeur assez élevée. Les variables du coup (3) $TA\beta \times y$ sont alors complètement connues : ce coup ne peut être que $Ta2 \times c2$.

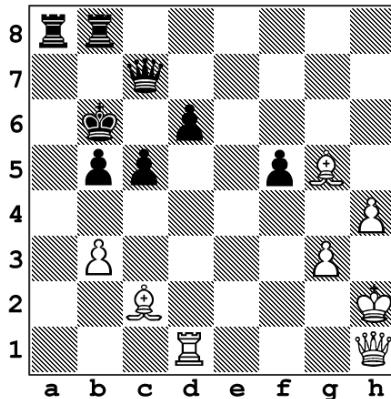


FIG. 9.11 – Utilisation d'un plan (trait aux noirs)

La tour est alors nécessairement initialement en *a8* et le coup (4) doit être *Ta8a2*. L'unification du schéma avec la position est donc réussie et le plan donné par l'arbre instancié est essayé. Pour voir s'il peut l'être avec succès, le programme appelle ensuite tous les plans de ripostes noires possibles tels qu'ils sont eux-mêmes stockés dans les schémas. Si le résultat final, évalué par le *minmax* est positif pour le camp ami, le plan de départ est lancé sinon les autres plans sont essayés au premier niveau. Cette procédure a été reprise dans un autre programme de J. Pitrat déjà étudié en 6.3.

Conclusion

Le programme n'apprend que lorsqu'il a compris. Il est indifférent aux mauvais coups joués : si le bon coup n'a pas été trouvé dans la partie réelle, il n'apprend rien ; si un coup meilleur existait, peu importe. Il a appris quand même quelque chose. Ce qu'il apprend dépend de ce qu'il sait puisque pour comprendre une nouvelle situation, il utilise les schémas déjà enregistrés. Il se peut donc qu'il ne comprenne pas un coup difficile à un certain stade de son développement, mais qu'il y parvienne plus tard après avoir joué d'autres parties .

Une autre caractéristique très importante de ce travail est que la démarche est valable pour beaucoup d'autres jeux. Les plans appris sont très souples ; en outre, contrairement au système de Waterman. Il ne sont pas impératifs mais donnent seulement une **indication des bons coups possibles** : des facteurs extérieurs minlmax ou stratégie à long terme décideront.

En revanche, une sérieuse difficulté subsiste : le nombre de schémas de plans engendrés est immense (de l'ordre de la dizaine de mille pour l'ensemble du jeu). Ceci est dû aux nombreuses variantes possibles pour une situation et à la multiplicité de ces situations aux échecs. De plus, il est très difficile d'éliminer, les coups étrangers des plans et de simplifier ces derniers suffisamment.

Ainsi, le programme apprend mais ne sait pas utiliser de manière efficace son nouveau savoir.

Ce problème fondamental n'est pas résolu actuellement. On voit que, curieusement, ce n'est pas un problème d'apprentissage , mais, « simplement », un problème de **gestion d'une base immense de connaissances**. On rejoint là les préoccupations signalées en conclusion des travaux sur les systèmes-experts et toute la problématique de la métacognition.

9.6 Apprentissage de caractéristiques

Un jeune enfant apprend à forger des concepts et à leur faire correspondre un nom : il se fait une idée de ce qu'est un homme, un jouet, un arbre, une maison... Comment faisons-nous pour reconnaître des exemples divers d'un même concept ? Comment apprenons-nous les caractéristiques de celui-ci ?

Patrick H. Winston a écrit un programme (1970) qui fournit des éléments de réponse à ces deux questions.

Steven A. Vere propose en 1979 un système plus puissant qui sera décrit ensuite .

Le programme de P. M Winston analyse, par caméra, des scènes composées d'objets simples : cubes, parallélépipèdes, prismes, pyramides. Chaque scène est d'abord traitée par un programme qui en extrait une représentation raisonnable : chaque objet est isolé et les relations entre objets sont mémorisées à l'aide d'un graphe étiqueté, dont les sommets sont les objets .

Les relations ainsi détectées sont :

AU-DESSUS-DE , A-COTE-DE, SUPPORTE-PAR , A-GAUCHE-DE, DEVANT et EST-UN-GENRE-DE.

La scène de la figure 9.12.a est représentée par le graphe de la figure 9.12.b.

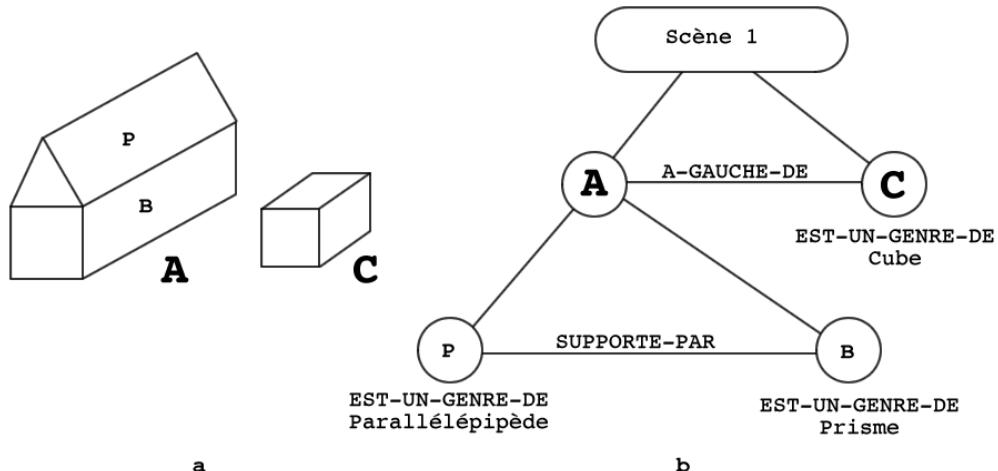


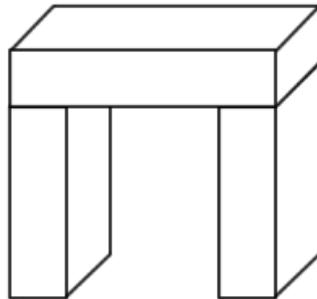
FIG. 9.12 – Analyse et représentation d'une scène

Le système, aidé par un professeur, est capable d'apprendre, à travers plusieurs exemples et contre-exemples, le concept discriminant de la scène. Toute l'expérience se rapproche un peu de la démarche par analogie nécessaire pour résoudre des scènes de tests de Quotient Intellectuel. Il s'agit de **trouver une loi** vérifiée par les premiers exemples. Nous avions vu au chapitre 3 comment une partie de ce travail pouvait être fait par l'algorithme d'unification, et comment le programme de Thomas Evans résolvait de tels tests.

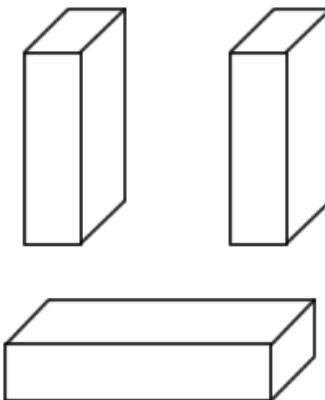
Dans le système de P. H. Winston, une succession de scènes commence par une situation typique que le système mémorise simplement dans sa représentation. Mais, et il y a là un changement de conditions d'expérience important, le professeur peut alors proposer au système des scènes qui ne sont pas des exemples corrects du concept à apprendre. Celui-ci va donc apprendre à la fois à partir des exemples et des **contre-exemples**.

Une idée cruciale est le « presque exemple » (« near-miss ») : si, en effet une situation radicalement différente de toutes les autres est soudain présentée, il n'y a rien à dire : si au contraire une situation presque semblable est soumise, le système va s'intéresser aux différences avec le modèle interne et marquer celles-ci, apprenant, petit à petit, les caractéristiques essentielles et secondaires du concept.

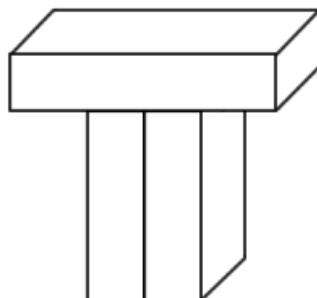
Supposons que le professeur présente d'abord la scène :



Le système la décrit comme composée de deux briques debout qui supportent une brique couchée. Si la scène, donnée comme contre-exemple, est :



Le système va marquer dans le graphe la relation de support indispensable dans le concept. Lorsque la troisième scène, qui est à nouveau un contre-exemple, apparaît :



Le programme ajoute cette fois à la représentation interne la relation « NON-A-COTE-DE » entre les deux briques debout. Si, par la suite, un exemple est donné où la brique couchée devient un prisme, une généralisation est faite pour cet objet qui devient toute forme de base plane (de façon à pouvoir être supportée par deux briques). Finalement le concept « arche » aura été appris sous la forme de la figure 9.13 :

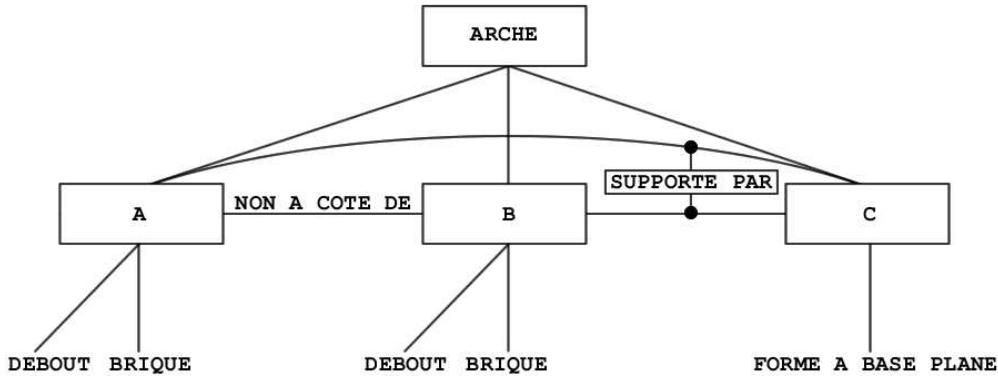


FIG. 9.13 – Le concept d’arche

Si en face d'une situation donnée, il existe pour le système, plusieurs façons de faire correspondre la scène courante au modèle, c'est une fonction d'évaluation qui, établissant une mesure de priorité entre les objets et entre les relations indique quelle action entreprendre pour chaque différence possible. Les limites de ce programme tiennent à deux aspects :

Le premier est la pauvreté de l'univers de travail : les relations concernent seulement les objets deux à deux, elles sont en très petit nombre, comme les objets eux-mêmes ; les contre-exemples sont naïfs, et rien n'indique comment « bien les choisir ».

Le second aspect est le côté uniquement empirique de l'apprentissage : aucune convergence n'est assurée ; comment faire, d'autre part, pour reconnaître un objet si, non pas un, mais mille modèles sont disponibles ?

A la limite, le système ne peut en réalité converger, et tous les modèles sont équivalents : en effet, le système ne retient que le dernier modèle et non les exemples qui lui ont donné naissance ; Il n'a pas de possibilité de revenir en arrière. Lorsqu'il ajoute, par exemple la relation NON-A-COTE-DE, il n'a pas la possibilité de tester qu'effectivement celle-ci était vérifiée dans tous les exemples précédents. C'est sur les autres possibilités, laissées de côté par la fonction d'évaluation, qu'il faudrait pouvoir revenir dès qu'une contradiction est trouvée.

Le travail de Stephen A .Vers (1979) est un progrès dans ces deux voies.

9.7 Apprentissage de concepts

L'apprentissage consiste essentiellement à abstraire, par induction à partir d'un petit nombre d'exemples, une situation générale. Cet apprentissage peut être guidé et accéléré par des contre-exemples, qui seront ici absolument quelconques et non plus seulement des « presque exemples ». Les situations rencontrées peuvent être complexes et requérir des représentations comportant plusieurs niveaux emboîtés avec des disjonctions et des négations. Cette représentation permet en particulier de garder en réserve toutes les éventualités de ressemblance et de différence entre situations. Portons dans un premier temps notre attention sur le mécanisme d'abstraction.

Abstraction et généralisation

L'unification habituelle a pour support la substitution de variables par

des termes. Ces substitutions sont de nature déductive et rigoureuse. **Généraliser** consiste précisément à marcher en sens inverse : **un terme est abstrait s'il est remplacé par une variable.** Une substitution de ce type, qu'on dira *inductive*, doit alors préciser quelle occurrence du terme doit être substituée puisque, contrairement à l'unification, aucune substitution n'est forcée. Rien n'impose, par exemple, qu'un même terme soit partout remplacé par la même variable.

Une telle substitution inductive O est donc caractérisée par une liste de triplets : (terme, occurrences / variable substituée) et sera notée :

$$O = \{(t_1, o_1/v_1); \dots; (t_n, o_n/v_n)\}$$

Si l'on a l'expression E :

$$E = ((\text{pièce de})\text{Voltaire Candide}) \wedge (\text{français Voltaire}) \wedge (\text{écrivain Voltaire})$$

La substitution déductive : $O = \{(\text{Voltaire}, (1, 3)/x); (\text{Candide}, 1/y)\}$ transforme E en OE :

$$OE = ((\text{pièce de})xy) \wedge (\text{français Voltaire}) \wedge (\text{écrivain } x))$$

Notons donc que si l'on ne prend pas plus de précaution, il n'existe pas nécessairement de substitution déductive, D , qui permette à partir d'une expression généralisée, de retrouver l'original. Il suffit par exemple ci-dessus de changer également le terme Candide en x . Comme le retour *aux* exemples originaux est requis pour de simples raisons de cohérence du système, on interdira désormais dans les substitutions inductives :

- a) **qu'à deux objets différents soit substituée la même variable**
- b) **qu'une substitution réutilise des variables de l'expression qu'on généralise**

Moyennant cette définition et ces restrictions, nous sommes en possession d'un mécanisme de généralisation : Une expression E_1 est dite *plus générale* qu'une expression E_2 , si et seulement si il existe une substitution inductive O telle que : $E_1 \subseteq OE_2$.

où, par convention, E_1 et OE_2 ont été normalisées sous formes normales conjonctives et où l'inclusion doit s'entendre au sens ensembliste : chaque facteur de E_1 apparaît dans OE_2 . Dans ces expressions, l'ordre des facteurs n'intervient en aucune façon. Ainsi, l'expression E_1 :

$$E_1 = (x\spadesuit) \wedge (y\heartsuit) \wedge (x \geq y)$$

est plus générale que E_2 :

$$E_2 = (As\spadesuit) \wedge (R\spadesuit) \wedge (10\spadesuit) \wedge (R\heartsuit) \wedge (D\heartsuit) \wedge (As \geq R) \wedge (R \geq D)$$

puisque il existe :

$$O = \{AS, 1, 2/x, R(1, 2, 3, 4)/y\}$$

telle que $E_1 \subseteq OE_2$.

On écrira dorénavant : $E_1 \leq E_2$

pour signifier que E_1 est plus générale que E_2 et on dira alors aussi que E_1 *couvre* E_2 . Pour E_1 , E_2 et O données, on appelle *couplage* la partie de OE_2 qui est dans E_1 et *résidu* la partie complémentaire.

Dans l'exemple ci-dessus, le couplage est donc :

$$(x\spadesuit) \wedge (y\heartsuit) \wedge (x \geq y)$$

tandis que le résidu est :

$$(y\spadesuit) \wedge (10\spadesuit) \wedge (D\heartsuit) \wedge (y \geq D)$$

Bien entendu, pour E_1 et E_2 données, il peut exister plusieurs substitutions O_i qui vérifient :

$$E_1 \leq O_i E_2$$

Elles peuvent, en particulier, conduire à des couplages identiques et des résidus différents.

On peut songer à plusieurs politiques pour définir ces substitutions inductives : conservatives si le changement minimum est requis, libérales si toute substitution est autorisée. Dans la suite nous retiendrons les seules substitutions *équitables*, c'est à dire, celles où, en plus des premières restrictions a) et b) on a :

- c) **un même terme se trouve substitué dans toutes ses occurrences par la même variable,**
- d) **un terme n'est substitué que si c'est nécessaire ; en particulier un terme du résidu ne peut être remplacé que s'il l'a déjà été dans le couplage.**

Il n'y a plus lieu, dès lors, d'indiquer les occurrences des termes dans les substitutions équitables.

Cette définition entraîne que toute expression est sa propre généralisée et aussi celle de toute expression déduite d'elle même par simple renommage des variables. Une généralisation E_1 de E_2 sera au contraire stricte si et seulement si :

$\exists O$ avec $E_1 < OE_2$ (strictement) ou bien $E_1 = OE_2$ à condition que $O \neq$ renommage. La relation $<$ implique dès lors la relation \leq , mais non l'inverse. Comme on définit le plus grand commun diviseur en arithmétique, ou encore le maxterme en algèbre booléenne, la *généralisation commune maximale* $E.gcm$ en abrégé, de deux expressions E_1 et E_2 , est ici définie par :

$$\left\{ \begin{array}{l} E \leq E_1 \\ E \leq E_2 \end{array} \right. \text{ et } \nexists E' \text{ telle que } \left\{ \begin{array}{l} E' < E \leq E_1 \\ E' < E \leq E_2 \end{array} \right.$$

Un concept est alors une généralisation commune maximale de ses exemples. Les exemples sont considérés deux à deux ; tous les couplages possibles sont systématiquement retenus et on itère jusqu'à obtenir tous les sous-ensembles disjoints d'expressions généralisant tous les exemples fournis. La gcm finale comporte donc éventuellement des disjonctions. L'introduction de contre-exemples dans ce processus va à la fois affiner et compliquer cette analyse.

Les contre-exemples apportent bien évidemment des négations dans la description du concept. Celles-ci peuvent en outre être mélangées à des disjonctions, le tout étant imbriqué de façon quelconque et les généralisations finales vont devenir passablement complexes.

Par exemple :

$$(\text{SUR } x \ y) \wedge (\text{rouge } y) \wedge \neg(((\text{SUR } z \ y) \wedge (\text{cube } z)) \vee (\text{cube } y))$$

« Quelque chose se trouve sur un objet rouge et il n'y a pas d'objet qui n'est pas un cube sur cet objet rouge ou bien alors y lui-même n'est pas un cube ».

Remarquons que, dans ces descriptions, la notation :

$$A \wedge \neg(B \wedge \neg C)$$

est préférée à la notation équivalente :

$$(A \wedge \neg B) \vee (A \wedge B \wedge C)$$

à cause de la simplicité du processus d'apprentissage lié à la première notation.

Le système reçoit en une seule fois une famille d'exemples p_i pour $i = 1$ à p , ainsi qu'une famille de contre-exemples v_j , pour $j = 1$ à q . Il s'intéresse d'abord à la famille des exemples. Il en bâtit la gcm P_1 . Celle-ci est a priori trop large : elle permet aussi de décrire certains contre-exemples ; l'examen de ceux-ci introduit un terme correctif N_1 qui ne doit donc pas être rencontré dans le concept. Dans l'expression C_1 qui décrit le concept qui doit être appris, N_1 est une condition négative :

$$C_1 = P_1 \wedge \neg N_1$$

Mais le terme $\neg N_1$ peut maintenant corriger P_1 un peu trop fort et éliminer ainsi certains exemples. Une nouvelle condition négative (« counterfactual ») est ajoutée et ainsi de suite. Comme les formules descriptives de départ ainsi que le nombre total de cas sont finis, le processus est assuré de parvenir au bout du compte à une expression ultime elle-même finie ; dans le pire des cas, celle-ci sera la disjonction de tous les cas rencontrés. En général, E convergera vers une expression plus compacte, C , qui exprimera concisément les caractéristiques que partagent les exemples et qui ne figurent pas dans les contre-exemples, c'est à dire, décrira le concept à apprendre. La forme générale de C sera :

$$C = P_1 \wedge \neg(N_1 \wedge \neg N_2 \wedge \neg(\dots \wedge \neg N_k))$$

L'analogie avec les séries alternées convergentes de l'analyse classique, de la forme :

$$a_0 - A_1 x^1 + a_2 x^2 + \dots + (-1)^k a_k x^k$$

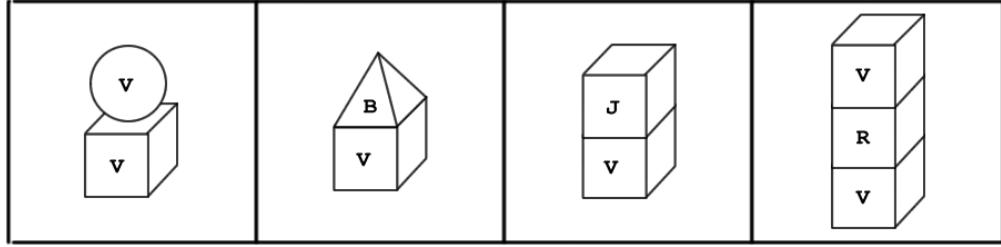
s'impose immédiatement. Les termes successifs sont simplement des propositions symboliques au lieu d'être des monômes à valeurs numériques.

Illustrons à l'aide d'un exemple cette procédure de construction de la forme C dite généralisation commune maximale. Le système reçoit les descriptions p_1, p_2, p_3 et p_4 et v_1, v_2, v_3 et v_4 des quatre exemples et des quatre contre-exemples de la figure 9.14.

Exemples :

Descriptions

$$p_1 = (O_1 \text{ SUR } O_2) \wedge (\text{SPHERE } O_1) \wedge (\text{CUBE } O_2) \wedge (\text{VERT } O_1) \wedge (\text{VERT } O_2)$$



$$p_2 = (O_3 \text{ SUR } O_4) \wedge (\text{PYRAM } O_3) \wedge (\text{CUBE } O_4) \wedge (\text{BLEU } O_3) \wedge (\text{VERT } O_4)$$

$$p_3 = (O_5 \text{ SUR } O_6) \wedge (\text{CUBE } O_5) \wedge (\text{CUBE } O_6) \wedge (\text{JAUNE } O_5) \wedge (\text{VERT } O_6)$$

$$p_4 = (O_7 \text{ SUR } O_8) \wedge (O_9 \text{ SUR } O_7) \wedge (\text{CUBE } O_7) \wedge (\text{CUBE } O_8) \wedge (\text{CUBE } O_9) \wedge (\text{ROUGE } O_7) \wedge (\text{VERT } O_8) \wedge (\text{VERT } O_9)$$

Contre-exemples :

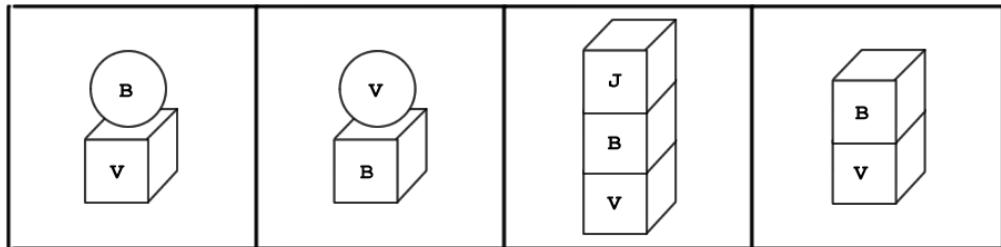


FIG. 9.14 – Exemples et contre-exemples d'un concept

$$v_1 = (Q_1 \text{ SUR } Q_2) \wedge (\text{SPHERE } Q_1) \wedge (\text{CUBE } Q_2) \wedge (\text{BLEU } Q_1) \wedge (\text{VERT } Q_2)$$

$$v_2 = (Q_3 \text{ SUR } Q_4) \wedge (\text{SPHERE } Q_3) \wedge (\text{CUBE } Q_4) \wedge (\text{VERT } Q_3) \wedge (\text{BLEU } Q_4)$$

$$v_3 = (Q_5 \text{ SUR } Q_6) \wedge (Q_7 \text{ SUR } Q_5) \wedge (\text{CUBE } Q_7) \wedge (\text{CUBE } Q_5) \wedge (\text{CUBE } Q_6) \wedge (\text{JAUNE } Q_7) \wedge (\text{BLEU } Q_5) \wedge (\text{VERT } Q_6)$$

$$v_4 = (Q_8 \text{ SUR } Q_9) \wedge (\text{CUBE } Q_8) \wedge (\text{CUBE } Q_9) \wedge (\text{BLEU } Q_8) \wedge (\text{VERT } Q_9)$$

La gcm pour les quatres exemples est P_1 :

$$P_1 = (x \text{ SUR } y) \wedge (\text{CUBE } y) \wedge (\text{VERT } y)$$

Le contre-exemple v_2 est éliminé par P_1 , car le cube de v_2 n'est pas VERT. Mais en revanche, v_1, v_3 et v_4 sont couverts : Il faut les éliminer en ajoutant un terme correctif $\neg N_1$.

Ce terme doit éliminer, par hypothèse, les résidus $p'_i = v_i - O_I P_1$ pour $i = 1, 3$ et 4 et les substitution O_i convenables :

$$p'_1 = (\text{SPHERE } x)(\text{BLEU } x) \quad \text{avec } O_1 = (Q_1/x, Q_2/x)$$

$$p'_3 = (Q_7 \text{ SUR } x)(\text{CUBE } Q_7)(\text{CUBE } x)(\text{JAUNE } Q_7)(\text{BLEU } x) \quad \text{avec } O_3 = (Q_5/x, Q_6/x)$$

$$p'_4 = (\text{CUBE } x)(\text{BLEU } x) \quad \text{avec } O_4 = (Q_8/x, Q_9/y)$$

L'expression N_1 doit couvrir ces trois nouveaux exemples ; elle prend la valeur de leur gcm qui est ici tout simplement :

$$\text{gcm}(p'_1, p'_3, p'_4) = N_1 = (\text{BLEU } x)$$

La description du concept cherché devient alors :

$$C_1 = (x \text{ SUR } y) \wedge (\text{CUBE } y) \wedge (\text{VERT } y) \wedge \neg(\text{BLEU } x)$$

L'expression C_1 couvre alors toujours bien les exemples initiaux p_1, p_3 et p_4 : aucun d'entre eux ne comporte en effet un facteur ($\text{BLEU } x$). Au contraire, ce facteur est présent dans p_2 qui est donc pour l'instant éliminé. Pour le réintroduire, les facteurs non couverts par un terme positif ou négatif de C_1 forment le nouveau résidu.

Il reste ici simplement dans p_2 : ($\text{PYRAM } x$) qui est sa propre gcm et fournit le dernier terme correctif.

La description apprise du concept défini par la figure 9.14 est donc finalement :

$$C = (x \text{ SUR } y) \wedge (\text{CUBE } y) \wedge (\text{VERT } y) \wedge \neg((\text{BLEU } x) \wedge \neg(\text{PYRAM } x))$$

Il se peut, bien sûr, que la procédure de génération des gcm fournisse à une étape donnée, non pas une, mais plusieurs expressions. Des disjonctions sont dans ce cas introduites dans la description. Il se peut également que la procédure ne fournisse aucun gcm, soit parce que les facteurs d'origine ne partagent aucune relation, soit à cause des restrictions imposées aux unifications. Ceci montre que la seule expression généralisée est vide, c'est-à-dire, qu'il n'existe pas de concept effectif couvrant les exemples et éliminant les contre-exemples. Ceci se produit par des contradictions entre résidus du type :

$$\text{résidu positif} = (x \text{ SUR } y)(\text{VERT } x), \text{ résidu négatif} = (x \text{ SUR } y)(\text{VERT } x)(\text{ROUGE } y)$$

Le programme de Vere est écrit dans le langage ICECUBE, un dialecte de SNOBOL et produit les généralisations en quelques secondes sur un IBM370/158. Il est aussi capable de généraliser des règles de réécriture décrivant, par exemple, des empilements-dépilement de cubes. S'il reste à compléter ce travail, en enrichissant notamment le vocabulaire des descriptions par d'autres relations, il se présente néanmoins comme l'une des recherches les mieux formalisées et les plus extensibles en apprentissage. Les concepts appris sont déjà d'une belle **complexité qui dépasse peut-être ce que l'homme est lui-même capable de former et mémoriser.**

Abréviations

AAAI	American Association for Artificial Intelligence.
ACM Computing Surveys : Revue de l'ACM.	
AFCET	Association Française de Cybernétique et d'Etudes Techniques.
AFIPS	American Federation for Information Processing Society.
A.I.	Artificial Intelligence, Revue, North Holland.
CACM	Communications of the Association for Computing Machinery.
CAD	Computer Aided Design.
CAO	Conception Assistée par Ordinateur
C.S.	Cognitive Science, Revue.
ECAI	European Conference on Artificial Intelligence.
IEEE	Institute for Electrical and Electronic Engineers.
IFIP	International Federation of Information Processing.
IJCAI	International Joint Conference on Artificial Intelligence.
JACM	Journal of the Association for Computing Machinery.
M.I.	Machine Intelligence, Recueil d'articles, 9 tomes, Elsevier.
M.I.T.	Massachusetts Institute of Technology, Cambridge USA.
OPN. RES.	Operations Research, Revue.
RAIRO	Revue d'Informatique et de Recherche Opérationnelle, Dunod.
SIGART	Special Interest Group in Artificial Intelligence (ACM).
TSI	Technique et Science Informatique, Revue, Dunod

Bibliographie

- [1] H. Kanoul A. Colmerauer and M. Van Caneghem. *Prolog, bases théoriques et développement actuels*. TSI, 1983.
- [2] A. Collins A. Stevens and S. Goldin. *Misconceptions in student's understanding*. International Journal of man-machine Studies, 1979.
- [3] R. P. Abelson. *Concepts for representing mundane reality in plans*. Bobrow and Collins, 1975.
- [4] J.R. Abrial. Data semantics. *IFIP, TC2*, 1974.
- [5] R. L. Ackoff. *The art of problem solving*. J. Wiley, 1978.
- [6] A. Adam. « *Gadget* » *Un programme de génération automatique de programmes sur les graphes et les ensembles*. Université Paris VI, 1973.
- [7] A. Adam. *Utilisation des transformations sémantiques pour la correction automatique des programmes*. Université Paris VI, 1978.
- [8] A. Adam and J. P. Laurent. *LAURA, A system to debug students programs*. A.I., 1980.
- [9] L. Aillo and G. Levi. The uses of metaknowledge in ai systems. *ECAI*, pages 705–719, 1984.
- [10] J. P. Alem. *Jeux de l'esprit et divertissements mathématiques*. Seuil, 1975.
- [11] J. J. Allan. *CAD systems*. North Holland, 1976.
- [12] S. Amarel. *On representations of problems of reasoning about actions*. Fisovier, 1968.
- [13] S. Amarel. *On representations of problems of reasoning about actions*. Fisovier, 1968.
- [14] S. Amarel. *On representations of problems of reasoning about actions*. Fisovier, 1968.
- [15] J. Anderson. *Induction of augmented transition networks*. C. S., 1977.
- [16] J. Anderson and P. Kline. *A Learning system and its psychological implications*. IJCAI, 1979.
- [17] J. R. Anderson. *Language, memory and thought*. Lawrence Eribaum associates, 1976.
- [18] J. R. Anderson and G. Bower. *Human associative memory*. Winston-Wiley, 1973.
- [19] R. Anderson and J. Gillogly. Rand intelligent terminal agent rita design and philosophy. *R-1809-ARPA*, 1976.
- [20] R. Anderson, P. Kline, and C. Beasley. *Aptitude Learning and Instructions*, chapter Complex learning processes. 1979.

-
- [21] Asia-pacific conference on operational research. *Predictive complicate coordinating plannings*, Singapore, 1982.
 - [22] J. Atkins. Prototypes and production rules : and approach to knowledge representation for hypothesis formation. *IJCAI*, 6, 1979.
 - [23] M. Ballantyne. *Computer generation of counter example in topology*. PhD thesis, Univ. Texas, Austin, 1975.
 - [24] R. Banerji. *A.I. a theoretical approach*. North Holland, 1980.
 - [25] R. Banerji and G. W. Ernst. Strategy construction using homomorphism between games. *A.I.*, 3,4, 1972.
 - [26] Y. Bar-Hillel. *Language and informations*. Addison-Wesley, 1964.
 - [27] J. Barnett and M. Bernstein. Knowledge-based systems : a tutorial. *System Development Corporation*, TM(L) 5903, 1977.
 - [28] A. Barr and E. A. Feigenbaum. *Le Manuel de l'Intelligence Artificielle*. Eyrolles, 1986.
 - [29] D. Barstow. *Knowledge based program construction*. Elsevier, 1979.
 - [30] D.R. Barstow. An experiment in knowledge-based automatic programming. *A.I.*, 12 :73–119, 1979.
 - [31] U. Bartels, W. Otthoff, and P. Rawlefs. Ape : a system for automatic programming from abstract specifications of data types and algorithms. *IJCAI*, 7 :1037–1043, 1981.
 - [32] F. Bartlett. *Remembering : A study in experimental and social psychology*. PhD thesis, Cambridge University, 1932.
 - [33] F. Bartlett. *Thinking*. Basic Books, 1958.
 - [34] G. Battani and H. Meloni. Un interpréteur de prolog. *GIA Luminy*, 1973.
 - [35] G. Baudet. *On the branching factor of the alpha-beta pruning algorithm*. *A.I.*, 1978.
 - [36] J.S. Bennett and R. Engelmore. Sacon : a knowledge-based consultant for structural analysis. *IJCAI*, 6 :47–49, 1979.
 - [37] J.S. Bennett and C.R. Hollander. Dart : an expert system for computer fault diagnosis. *IJCAI*, 6 :843–845, 1981.
 - [38] O. Berge. *Graphes et hypergraphes*. Dunod, 1970.
 - [39] H. Berliner. *The B* tree search algorith : a best proof procedure*. *A.I.*, 1979.
 - [40] H. Berliner. *Backgammon computer program beats world champion*. *A.I.*, 1980.
 - [41] H. Berliner. *Computer backgammon*. Scientific American, 1980.
 - [42] P. Berloquin. *Jeux mathématiques du monde*. Flammarion, 1978.
 - [43] P. Berloquin. *100 Jeux numériques - 100 Jeux géométriques pour insomniaques*. Livre de Poche, 1980.
 - [44] W. Bibel. *Syntax-directed, semantic-supported program synthesis*. *A.I.*, 1980.
 - [45] W. Bibel. *A comparative study of several proof procedures*. *A.I.*, 1982.
 - [46] J. Bitner and E. Reingold. *Backtrack programming techniques*. ACM, 1976.
 - [47] J. M. Blanchet. *Mathématiques en liberté*. OCDL, 1976.
 - [48] W. Bledsoe. Non-resolution theorem proving. *A.I.*, 9, 1977.
 - [49] W. Bledsoe. Non-resolution theorem proving. *A.I.*, 9, 1977.
 - [50] W. W. Bledsoe. *Splitting and reduction heuristics in automatic theorem proving*. *A.I.*, 1971.

- [51] W. W. Bledsoe and P. Bruel. *A man-machine theorem proving system*. IJCAI, 1973.
- [52] D. Bobrow. *Representation and understanding*. Academic Press, 1975.
- [53] D. Bobrow. *Representation and understanding*. Academic Press, 1975.
- [54] D. Bobrow and T. Winograd. Krl another perspective. *CS*, I(1) :24–79, 1979.
- [55] D. G. Bobrow. *Natural language input for a computer problem solving system*. PhD thesis, MIT, 1968.
- [56] D. G. Bobrow. *Special issue on non-monotonic logic*. A.I., 1980.
- [57] D. J. Bobrow and T. Winograd. *A knowledge representation language*. C. S., 1977.
- [58] M. Boden. *AI and natural man*. Basic Books, 1977.
- [59] N. Bongard. *Pattern recognition*. Spartan Books, 1970.
- [60] A. Bonnet. *Analyse de textes au moyen d'une grammaire sémantique de schémas. Applications à la compréhension de résumés médicaux en langage naturel*. PhD thesis, Paris VI, 1980.
- [61] A. Bonnet. *Analyse de textes au moyen d'une grammaire sémantique et de schémas. Application à la compréhension de résumés médicaux en langage naturel*. PhD thesis, Université Paris VI, 1980.
- [62] A. Bonnet. Applications de l'intelligence artificielle, les systèmes experts. *RAIRO Informatique*, 15 (4), 1981.
- [63] A. Bonnet. *L'intelligence artificielle, Promesses et Réalités*. Interéditions, 1984.
- [64] A. Bonnet, M.O. Cordier, and D. Kayser. An icai system for teaching derivatives in mathematics. *Proc. of 3rd World Conference on Computer Education (WCCE)*, pages 27–31, 1981.
- [65] A. Bonnet and C. Dahan. Oil-well date interpretation using expert system and pattern recognition technique. *IJCAI*, 8 :185–189, 1983.
- [66] A. Bonnet, J. Harry, and J.G. Ganascia. Litho, un système expert inférant la géologie du sous-sol. *TSI*, 1 (5) :393–402, 1982.
- [67] J.H. Boose. Personnal construct theory and the transfer of human expertise. *ECAI*, pages 51–60, 1984.
- [68] M. Borillo and J. Virbel. *Analyse et validation dans l'étude des données textuelles*. 1977.
- [69] A. Borning and A. Bundy. Using matching in algebraic equation solving. *IJCAI*, 7 :466–471, 1981.
- [70] M. Botwinnink. *Computers chess and long range planning*. Springer Verlag, 1970.
- [71] D. Bourgoin. *PARI, un programme heuristique pour résoudre des exercices d'arithmétique*. PhD thesis, Paris VI, 1978.
- [72] D. Bourgoin. *PARI, un programme heuristique pour résoudre des exercices d'arithmétique*. PhD thesis, Paris VI, 1978.
- [73] R.J. Brachman. What's in a concept : Structural foundations for semantic networks. *International Journal of man-machine studies* 9, 2 :127–152, 1977.
- [74] M. Brady. Computational approach to image understanding, 1983.
- [75] J. Brown, R. Burton, and A. Boll. Sophie : a step towards creating a reactive learning environment. *International J. of Man-Machine Studies*, 7 :675–716, 1975.

-
- [76] J. Brown, R. Burton, and A. Boll. Sophie : a step towards creating a reactive learning environment. *International J. of Man-Machine Studies*, 7 :675–716, 1975.
 - [77] J. S. Brown and R. Burton. *Multiple representations of knowledge for tutorial reasoning*. Academic Press, 1975.
 - [78] J. S. Brown and R. Burton. *Diagnostic models for procedural bugs in basic mathematical skills*. CS 2, 1978.
 - [79] M. F. Brown. Doing arithmetic without diagrams. *A.I.*, 8, 1977.
 - [80] J. S. Bruner, J.J Goodnow, and G. A. Austin. *A study of thinking*. Wiley, 1956.
 - [81] B. Buchanan and F. Felgenbaum. Dendral and meta-dendral. *A.I.*, 11 :5–24, 1978.
 - [82] B. Buchanan and T. Mitchell. *Model directed learning of production rules*. 1979.
 - [83] B.G. Buchanan and R.O. Duda. Principle of rule-based systems. Technical report, Standford University, 1982.
 - [84] A. Bundy. Doing arithmetic with diagrams. *IJCAI*, 3, 1973.
 - [85] A. Bundy. *Artificial Intelligence*. Edinburgh University Press, 1981.
 - [86] A. Bundy, R. M. Burstall, S. Weir, and R. M. Young. *Artificial Intelligence : an introductory course*. Edinburgh University Press, 1980.
 - [87] A. Bundy, L. Byrd, G. Luger, R. Mellish, R. Milne, and M. Palmer. Mecho a program to solve mechanics problems. Technical Report 50, Dept of A.I. U. of Edinburgh, 1979.
 - [88] A. et al. Bundy. Solving mechanics problems using meta-level inference. *IJCAI*, 6 :1017–1027, 1979.
 - [89] M. Buthion. *Un programme qui résout formellement des problèmes de constructions géométriques*, volume 13. RAIRO Informatique, 1979.
 - [90] D. A. c Waterman. *Generalization learning techniques for automating the learning of heuristics*. A.I., 1970.
 - [91] J. Carbonell. Politic : automated ideological reasoning. *C.S.*, 2 :27–51, 1978.
 - [92] J. Carbonell. Politic : automated ideological reasoning. *C.S.*, 2 :27–51, 1978.
 - [93] J. Carbonell. *Counterplanning : a strategy-based model of adversary planning in real-world situations*. A.I., 1981.
 - [94] J.G. Carbonell. Towards a process model of human personality traits. *A.I.*, 15, 1980.
 - [95] R.E. Carhart. *CONGEN : on expert system aiding the structural chemist*. Edinburgh University Press, 1979.
 - [96] O. Carrière. *Réalisation d'un programme heuristique qui résout des tests psychologiques de mesure de facteur G*. Université Paris VI, 1973.
 - [97] J. Mc Carthy. *Epistemological problems in AI*. PhD thesis, MIT, 1977.
 - [98] J. Mc Carthy and P. Hayes. *Some philosophical problems from Standpoint of Artificial Intelligence*. Elsevier, 1969.
 - [99] E. Cauzinille. *Cognitive processes in the design of an intelligent tutoring system*. Colloque Tübingen, 1985.
 - [100] M. Cayrol. *Le langage LISP*. Cepadues éditions, 1983.
 - [101] M. Cayrol, B. Fade, and B. Fade. A decision-command function for a general robot : Argos-ii. Technical report, Université Toulouse, 1979.

- [102] M. Cayrol, B. Fade, and H. Farreny. Objets formels et attributs dans argos ii. *Actes Congrès AFCET*, pages 256–263, 1979.
- [103] J. Chailloux. *Le modèle VLISP description, implémentation et évaluation*. PhD thesis, Université de Vincennes, 1979.
- [104] J. Chailloux. *Le LISP, manuel de référence*. INRIA, 1985.
- [105] C. Chang and R. C. Lee. *Symbolic logic and mechanical theorem proving*. Academic Press, 1973.
- [106] C. L. Chang and J. R. Siagle. *Using rewriting rules for connection graphs to prove theorems*. A.I., 1979.
- [107] E. Charniak. *Computational semantics*. North Holland, 1976.
- [108] E. Charniak. On the use of frame knowledge in language comprehension. *A.I.*, 1, 1977.
- [109] E. Charniak. On the use of framed knowledge. *A.I.*, 11 (3) :225–266, 1978.
- [110] E. Charniak. *A.I. Programming*. Lawrence Erlbaum associates, 1980.
- [111] E. Charniak. A common representation for problem solving and language-comprehension information. *A.I.*, 16, 1981.
- [112] E. Charniak and D. Mc Dermott. *An introduction to Artificial Intelligence*. Addison-Wesley, 1984.
- [113] B. Chichetchi. *Compréhension du langage naturel. Traduction paraphrasée d'exercices sur le courant alternatif posés en persan*. PhD thesis, Université Paris VI, 1979.
- [114] N. Chomsky. *Syntactic structures*. Mouton la Hague, 1957.
- [115] N. Chomsky. *Aspects of the Theory of Syntax*. MIT Press, 1965.
- [116] N. Chomsky. *Aspects of the Theory of Syntax*. MIT Press, 1965.
- [117] N. Chomsky. *Rules and representation*. The behavioral and brain sciences, 1982.
- [118] E. Chouraqui. *Contributions à l'étude théorique de la représentation des connaissances*. PhD thesis, Nancy, 1982.
- [119] N. Chuquet. Jeux et esbatures. In *Jeux et esbatures*, number 1346. Bibl. Nationale, 1484.
- [120] W. J. Clancey. The epistemology of a rule-based expert system - a framework for explanation. *A.I.*, 20, 1983.
- [121] W.J. Clancey. Tutoring rules for guiding a case method dialog. *International J. of Man-Machine Studies*, 11 :25–49, 1979.
- [122] W.J. Clancey and R. Letsinger. Neomycin : reconfiguring a rule-based expert system for application to teaching. *IJCAI*, 7 :829–836, 1981.
- [123] M. Clarke. *Advances in computer chess*. Edinburgh U. Press, 1977.
- [124] W. F. Clocksin and Mellish. *Programmer en PROLOG*. Eyrolles, 1985.
- [125] W.F. Clocksin and C.S. Mellish. *Programming in PROLOG*. Springer-Verlag, Berlin, 1981.
- [126] H. Coelho. An introduction to the knowledge representation subfield. *ECAI*, pages 283–285, 1984.
- [127] H. Coelho, J.C. Cotta, and L.M. Pereira. How to solve it with prolog. *Lab. Nat. de Engenharia*, 1980.
- [128] K. Colby, S. Weber, and F. D. Hilf. *Artificial paranoia*. Pergamon Press, 1975.
- [129] N. Collins and D. Michie. *Machine Intelligence, Vol. 1*. American Elsevier, 1968.

-
- [130] A. Colmerauer. Programmation en logique du premier ordre. *Actes Journées La compréhension*, 1977.
 - [131] A. Colmerauer. *Metamorphosis grammar. Natural language communication with computers*. 1978.
 - [132] A. Colmerauer, H. Kanoul, R. Pasero, and P. Roussel. *Un système de communication homme-machine en français*. PhD thesis, Université d'Aix-Marseille, 1973.
 - [133] S. Cooks. *The complexity of theorem proving procedures*. 2th ACM sump. on theory of computing, 1971.
 - [134] M. O. Cordier and M. C. Rousset. Interactive operators in expert systems : Tango. *ECAI*, 1984.
 - [135] M.O. Cordier. *Commande d'un robot en langage naturel dans un domaine nécessitant des connaissances pramatiques : les recettes de cuisine*. PhD thesis, LRI, Parix XI, 1979.
 - [136] M.O. Cordier. *Commande d'un robot en langage naturel dans un domaine nécessitant des connaissances pramatiques : les recettes de cuisine*. PhD thesis, LRI, Parix XI, 1979.
 - [137] M.O. Cordier and M.C. Rousset. Propagation : another way for matching patterns in kbs. *Cyb. and Syst. Res. 2*, 1984.
 - [138] P. Mc Corduck. *Machines who think*. Freeman, 1979.
 - [139] D. Coulon and D. Kasyer. Un système de raisonnement à profondeur variable. *Congrès AFCET-TTY, Nancy*, pages 517–527, 1980.
 - [140] L. M. Pereira D. H. D. Warren and F. Pereira. *Prolog - The language and its implementation compared with LISP*. ACM, SIGPLAN Notice and SIGART Newsletter, 1977.
 - [141] E. Dale and D. Michie. *Machine Intelligence, Vol. 2*. American Elsevier, 1968.
 - [142] R. Dallard. *Présentation d'un programme de démonstration de théorèmes d'arithmétique*. Université Paris VI, 1974.
 - [143] R. Davis. Knowledge acquisition in rule-based systems : Knowledge about representations as a basis for system construction and maintenance. *D.A. Waterman and F. Hayes-Roth (Eds.)*, pages 99–134, 1978.
 - [144] R. Davis. Interactive transfer of expertise : acquisition of new inference rules. *MIT*, 11 :1–36, 1979.
 - [145] R. Davis. Interactive transfer of expertise : acquisition of new inference rules. *MIT*, 11 :1–36, 1979.
 - [146] R. Davis. Meta-rules : Reasoning about control. *A.I. 15*, 3 :179–222, 1980.
 - [147] R. Davis, H. Austin, I. Carlbow, B. Frawley, P. Pruchnik, R. Sneiderman, and J.A. Gilreath. The dipmeter advisor : interpretation of geological signals. *IJCAI*, 7 :846–849, 1981.
 - [148] R. Davis, B. Buchanan, and E. Shortliffe. Production rules as a representation for knowledge-based consultation program. *A.I.*, 8, 1977.
 - [149] R. Davis and K.J. King. An overview of production systems. *M.I.*, 8 :300–332, 1977.
 - [150] R. Davis and D.B. Lenat. *Knowledge Based Systems in Artificial Intelligence*. New York, McGraw-Hill, 1982.
 - [151] J. de Kleer, J. Doyle, G. J. Steele, and G. J. Sussman. *Amord : explicit control of reasoning*. Proc. Symp. Artificial Intelligence Programming languages, Sigplan notices, 1977.

- [152] J. de Kleer et al. Explicit control of reasoning. *Artificial intelligence : An MIT perspective*, 1 :93–116, 1979.
- [153] Bachet de Meziriac. *Problèmes paysans et délectables qui se font par les nombres*. A. Blanchard, Paris, 1612.
- [154] A. de Sainte-Lagüe. *Avec des nombres et des lignes*. Dunod.
- [155] D. de Werra. *A note on graph coloring*. Rairo R, 1974.
- [156] A. Deledicq. *Mathématiques buissonnières*. CEDIC, 1975.
- [157] J. L. Delhaye. *DATAL, un programme de démonstration automatique de théorèmes*. Université Paris VI, 1970.
- [158] D. Mc Dermott and J. Doyle. Non-monotonic logic i. *A.I.*, 13, 1980.
- [159] J. P. Desclés. Langages quasi-naturels et catégories grammaticales. In *Langages quasi-naturels et catégories grammaticales*. Actes du colloque : Domaine et objectifs de la recherche cognitive, 1982.
- [160] Y. Descottes and J.C. Latombe. Gari : a problem solver that plans how to machine mechanical parts. *IJCAI*, 7 :766–772, 1981.
- [161] M. Dincbas. Prolog et un exemple de programme expert écrit en prolog. *Doc. CERT*, 2/3122, 1979.
- [162] M. Dincbas. A knowledge-based expert system for automatic analysis and synthesis. *CAD, Proc. IFIP*, pages 705–710, 1980.
- [163] M. Dincbas. Le système de résolution de problèmes metalog. *CERT/DERI, Report 3146, Convention DRET 79, 1216*, 1980.
- [164] G. Doumeingts, D. Breuil, and L. Pun. *La gestion de production assistée par ordinateur*. Hermes Publishing, 1983.
- [165] J. Doyle. Tms : a true maintenance system. *A.I.*, 12 :231–272, 1979.
- [166] J. Doyle. *A truth maintenance system*. *A.I.*, 1979.
- [167] H. L. Dreyfus. *What computers cannot do*. Harper, 1979.
- [168] R.O. Duda. The prospector consultation system. Technical report, Final Report SRI 81 72, 1982.
- [169] R.O. Duda, J. Gaschning, and P. Hart. Model design in the prospector consultant system for mineral exploration. In *Michie 1980*, 1980.
- [170] R.O. Duda and J.G. Gaschning. Knowledge based expert systems come of age. *Byte*, 6(9) :238–283, 1981.
- [171] R.O. Duda, P.E. Hart, and G.L. Sutherland. Semantic network representations in rule-based inference systems. *D.A. Waterman and F. Hayes-Roth (Eds.)*, pages 203–222, 1978.
- [172] H. E. Dudeney. *536 Puzzles and curious problems*. Scribner's NY, 1967.
- [173] J. Nievergelt E. Reinfeld and N. Dec. *Combinatorial algorithms : theory and practice*. Prentice Hall, 1977.
- [174] C. Eastman. *Heuristic algorithms for automated space planning*. IJCAI, 1971.
- [175] C. Eastman. *Automatic space planning*. *A.I.*, 1973.
- [176] E. Elcock and A. Murray. *Experiments with a learning comment in a go moku playing program*. M.I., 1967.
- [177] R. Engelmore and A. Terry. Structure and function of the crysalys system. *IJCAI*, 6 :250–256, 1979.
- [178] Erii. *SAPHIR, présentation générale*. Rapport de la Société d'étude et de recherche en linguistique et informatique, 1983.

-
- [179] L. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy. The hearsay-ii speech-understanding system : integrating knowledge to resolve uncertainty. *Computing Surveys*, 12, 2, 1980.
 - [180] L. Erman and V. Lesser. A multi-level organization for problem-solving using many diverse cooperating sources of knowledge. *IJCAI*, 4 :483–490, 1975.
 - [181] L. Erman, P. London, and S. Fickas. The design and an example use of hearsay iii. *IJCAI*, 7 :409–415, 1981.
 - [182] G. Ernst and A. Newell. *GPS : a case study in generality and problem solving*. New York Academic Press, 1969.
 - [183] G. Ernst and A. Newell. *GPS : a case study in generality and problem solving*. New York Academic Press, 1969.
 - [184] G. W. Ernst. *The utility of independant subgoal in theorem proving*. Information and Control, 1971.
 - [185] G. W. Ernst and A. Newell. *GPS : a case study in generality and problem solving*. Academic Press, 1969.
 - [186] G. W. Ernst and A. Newell. *GPS : a case study in generality and problem solving*. Academic Press, 1969.
 - [187] T. G. Evans. *A heuristic program to solve geometric analogy problems*. PhD thesis, MIT, 1963.
 - [188] T. G. Evans. *A program for the solution of the geometric-analogy intelligence test questions*. MIT Press, 1968.
 - [189] Fahiman. *A planning system for robot construction tasks*. A.I., 1974.
 - [190] B. Failor. *Traitements inférents de faits conditionnels sur une application nécessitant un pattern-matching efficace : le jeu de la carte au bridge*. Université Paris Sud, 1985.
 - [191] B. Faller. *An expert system in symbolic integration*. ECAI, 1984.
 - [192] H. Farrency. *Un système pour l'expression et la résolution de problèmes orienté vers le contrôle de robots*. Université de Toulouse, 1980.
 - [193] H. Farrency. *Un système pour l'expression et la résolution de problèmes orienté vers le contrôle de robots*. PhD thesis, Toulouse, 1980.
 - [194] H. Farrency. Des puits de science qu'on peut sonder. *Science et Vie hors-série, La Science des Robots* :68–75, 1982.
 - [195] H. Farrency. *Programmer en LISP*. Masson, 1984.
 - [196] H. Farrency. *Les systèmes-experts, principe et exemples*. Cepadues, 1985.
 - [197] H. Farrency and R. Prajoux. Les pouvoirs des robots de la 3e génération. *Science et Vie*, 1982.
 - [198] E. Feigenbaum. *Artificial Intelligence : Themes in the Second Decade*. Information Processing 68. North Holland, 1968.
 - [199] E. Feigenbaum. The art of artificial intelligence. *IJCAI* 5, 5 :1014–1029, 1977.
 - [200] E. Feigenbaum, B. Buchanan, and J. Lederberg. On generality and problem solving : a case study using the dendral program. *M.I.*, 6 :165–190, 1971.
 - [201] E. Feigenbaum and P. Mc Corduck. *The fifth generation*. Addison-Wesley, 1983.
 - [202] E. Feigenbaum and J. Feldman. *Computers and Thought*. McGraw-Hill Book Company, 1963.
 - [203] J. Ferber. Mering : An open-ended object oriented language for knowledge representation. *ECAI*, pages 195–204, 1984.

- [204] P. Ferrand. Sesam : An explanatory medical aid system. *ECAI*, pages 13–20, 1983.
- [205] M. Fieschi. *Aide à la décision en médecine : le système Sphynx. Application au diagnostic d'une douleur épigastrique*. PhD thesis, Médecine, Marseille, 1981.
- [206] M. Fieschi. *Intelligence artificielle en médecine : systèmes experts*. Masson, 1984.
- [207] R. E. Fikes. *A system for solving problems stated as procedures*. A.I., 1970.
- [208] R. E. Fikes and N. J. Nilsson. *STRIPS : a new approach to the application of theorem proving in problem solving*. A.I., 1971.
- [209] C. Fillmore. *Universals in Linguistic theory*, chapter The case for case. Holt, Rinehart and Winston, 1968.
- [210] N. V. Findler. *Associative networks : the representation and use of knowledge by computers*. Academic Press, New York, 1979.
- [211] R. A. Finkel and J. P. Fishburn. *Parallelism in Alpha-Beta search*. A. I., 1982.
- [212] B. Flavigny. *Sur la détection a priori des erreurs dans les programmes*. Université Paris VI, 1972.
- [213] H. Floyd. *Non deterministic algorithms*. J.ACM., 1967.
- [214] C. Forgy and J. McDermott. Ops : a domain independent production system. *IJCAI*, 5 :933–939, 1981.
- [215] J. M. Fouet. *Conception par ordinateur de mécanismes à une boucle*. Université Paris VI, 1979.
- [216] J.M. Fouet. Désapprendre à programmer. *Publications du GR* 22, 30, 1982.
- [217] P. W. Frey. *Chess skill in man and machine*. Springer-Verlag, 1977.
- [218] L. Friedman. Extended plausible inference. *IJCAI*, 7 :487–495, 1981.
- [219] V. Arlazarov G. Adelson-Velskiy and M. Donskoy. *Some methods of controlling the tree in chess programs*. A.I., 1975.
- [220] H. Gallaire. Le langage prolog. 1981.
- [221] H. Gallaire and C. Lasserre. Controlling knowledge deduction in a declarative approach. *IJCAI*, 6 :S1–S12, 1979.
- [222] H. Gallaire and C. Lasserre. Controlling knowledge in a declarative approach. *IJCAI*, 6 :S1–S12, 1979.
- [223] H. Gallaire and J. Minker. *Logic and Data Bases*. Plenum, 1978.
- [224] J.C. Ganascia. Reasoning and result in expert systems : Main differences between diagnostic systems and problem-solvers. *ECAI*, pages 31–40, 1984.
- [225] J.G. Ganascia. *MIRLITHO : validation des résultats et détection des contradictions dans les systèmes de diagnostic*. PhD thesis, Paris Sud, 1983.
- [226] M. Gardner. *Les casses-têtes mathématiques de Sam Loyd*. Dunod, 1977.
- [227] M. Gardner. *L'effet haha*. Pour la science. Belin, 1979.
- [228] M. Gardner. *Les distracts. Les jeux mathématiques du Scientific American*. CE-DIC, 1979.
- [229] R. Garijo. *GPFA2, un système d'écriture automatique de programmes pour le calcul optimisé des fonctions récursives*. Université Paris VI, 1978.
- [230] O. Gascuel. Un programme général d'aide à la décision médicale structurant automatiquement ses connaissances. *Congrès AFCET RF-IA*, 1981.
- [231] P. Gaspar. *Problèmes, méthodes et stratégies de résolution*. Ed. Organisation, 1978.

-
- [232] H. Gelenier. *Realization of a geometry theorem proving machine*. Mc Graw-Hill, 1963.
 - [233] H. Gelernier. *Realization of a geometry theorem proving machine*. McGraw-Hill Book Company, 1963.
 - [234] D. Gelperin. *On the optimality of A**. A.I., 1977.
 - [235] M. Georgreff. A framework for control in production systems. *IJCAI*, 6 :328–334, 1979.
 - [236] D. Gerii and G. Girard. *Les olympiades internationales de mathématiques*. Hachette, 1976.
 - [237] M. Germain. Journées sur les spécifications. *AFCET-TTI Conference*, 1981.
 - [238] M. Ghallab. Decision trees for optimizing pattern-matching algorithm. *IJCAI*, 7 :310–312, 1981.
 - [239] J. P. Gilford. *The nature of human intelligence*. McGraw-Hill Book Company, 1967.
 - [240] N. Gillet. *Un exemple d'utilisation de connaissances en démonstration automatique*. Université Paris VI, 1980.
 - [241] P. C. Gilmore. An examination of the geometry theorem proving machine. *A.I.*, 1,3, 1970.
 - [242] P. Gloess. *GENER un générateur de programmes*. Université Paris VI, 1976.
 - [243] I. Goldstein and R. Roberts. Nudge : a knowledge-based scheduling program. *IJCAI*, 5 :257–263, 1977.
 - [244] I.P. Goldstein and E. Grimson. Annotated production system : A model for skill acquisition. *IJCAI*, 5 :311–317, 1977.
 - [245] S. Golomb and L. Baumert. *Backtrack programming*. J. ACM., 1965.
 - [246] M. Gondran and M. Minoux. *Graphes et algorithmes*. Eyrolles, 1979.
 - [247] M. Grandbastien. *Un programme qui résout formellement des équations trigonométriques par des procédés heuristiques*. Université Paris VI, 1974.
 - [248] C. Green. *Theorem-proving by resolution as a basis for question-answering systems*. Elsevier, 1969.
 - [249] C. Green. The design of the psi program synthesis system. *Proc. of the Second International Conference on software engineering*, pages 4–18, 1976.
 - [250] R. D. Greenblatt. *The Greenblatt chess program*, volume 31. Proc. AFIPS Fall Joint Computer Conference, 1967.
 - [251] A. De Groot. *Thought and choice in chess*. Mouton, 1965.
 - [252] M. Gross. *Grammaire transformationnelle du français : syntaxe du verbe*. Larousse, 1975.
 - [253] M. Gross. *Grammaire transformationnelle du français : syntaxe du nom*. Larousse, 1977.
 - [254] Mathlab Group. The macsyma reference manual. Technical report, MIT Lab. for Computer Science.
 - [255] A. Guzman. *Computer recognition of three-dimensional objects in a visual scene*. PhD thesis, MIT, 1968.
 - [256] A. Guzman. *Computer recognition of three-dimensional objects in a visual scene*. PhD thesis, MIT, 1968.
 - [257] A. Guzman. *Decomposition of the visual scene into three-dimensional bodies*. AFIP Fall Joint Conferences, 1968.

- [258] A. Guzman. *Decomposition of the visual scene into three-dimensional bodies.* AFIP Fall Joint Conferences, 1968.
- [259] C. Hagert. *Introduction to the learning subfield.* ECAI, 1984.
- [260] R. M. Haralick and G. L. Elliott. *Increasing tree search efficiency for constraint satisfaction problems.* A.I., 1980.
- [261] L. Harris. Robot a high performance natural language data base query system. *IJCAI*, 5, 1977.
- [262] P. Hayes. *Artificial and human thinking*, chapter The frame problem and related problems in A.I. Elsevier, 1973.
- [263] P. Hayes. *Artificial and human thinking*, chapter The frame problem and related problems in A.I. Elsevier, 1973.
- [264] P. Hayes. In defense of logic. *IJCAI*, 5 :559–583, 1977.
- [265] P. Hayes. On semantic nets, frames and associations. *IJCAI*, 5 :99–107, 1977.
- [266] F. Hayes-Roth. *Building expert systems.* Addison Wesley, 1983.
- [267] F. Hayes-Roth, P. Klahr, and D. Mostow. *Advice-taking and knowledge-refinement : An iterative view of skill acquisition.* Lawrence Erlbaum, 1981.
- [268] F. Hayes-Roth, D.A. Waterman, and D.B. Lenat. *Principles of pattern-directed inference systems.* Academic Press, 1978.
- [269] A. C. Hearn. *Reduce : a system and language for algebraic manipulation.* Peinick ed., 1971.
- [270] C.L. Hedrick. Learning production systems from examples. *A.I.*, 7 (1) :21–49, 1976.
- [271] G. G. Hendrix. *Partitionned networks for modeling natural language semantics.* PhD thesis, Univ. Texas, Austin, 1976.
- [272] J. Herbrand. *Une méthode de démonstration.* PhD thesis, Paris, 1931.
- [273] A. Hertz. *Programmes de démonstration de théorèmes formulables en logique des prédictats du premier ordre avec égalité.* Université Paris VI, 1975.
- [274] C. Hewitt. *PLANNER : A language for proving theorems in robots.* IJCAI, 1969.
- [275] C. Hewitt. *Description and theoretical analysis (using schemata) of PLANNER : A language for proving theorems and manipulating models in a robot.* Al Laboratory, M.I.T., 1971.
- [276] C. Hewitt. *Description and theoretical analysis (using schemata) of a PLANNER : a language for proving theorems.* PhD thesis, MIT, A.I. Lab, 1972.
- [277] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. *IJCAI*, 3, 1973.
- [278] D. R. Hofstadter. *Göedel, Escher et Bach.* Basic Books, 1977.
- [279] D. R. Hofstadter and D. G. Donnet. *The mind's I.* Basic Books, 1981.
- [280] A.W. Holt. *Introduction to Occurrence Systems.* Elsevier, 1971.
- [281] G. Huet. *Constrained resolution : A complete method for higher order logic.* Case Western Reserve University, 1972.
- [282] G. Huet. *A mechanization of type theory.* IJCAI 3, 1973.
- [283] G. Huet. *A Unification Algorithm for Type Theory.* Iria Laboria, 1973.
- [284] G. Huet. *A unification algorithm for typed lambda calculus.* Theoretical Computer Science 1, 1975.
- [285] G. Huet. *Unification dans les logiques d'ordre 1, 2, 3, oméga.* PhD thesis, Paris, 1978.

-
- [286] E. B. Hunt. *Artificial Intelligence*. Academic Press, New York, 1975.
 - [287] O. Itzinger. *Methoden der maschinellen Intelligenz*. Springer Verlag, 1976.
 - [288] P. Kline J. Anderson and C. Beasley. *Complex learning processes*. R. Snow ed., 1979.
 - [289] D. Sweeney J. D. Little, K. Murphy and O. Karel. *An algorithm or the travelling salesman problem*. Opn. Res., 1963.
 - [290] P. C. Jackson. *Introduction to Artificial Intelligence*. Petrocelli / Charter, 1974.
 - [291] T. Johnson. *The commercial applications of expert system technologies*. OVUM, 1984.
 - [292] G. Kahn. On when diagnostic systems want to do without causal knowledge. *ECAI*, 1984.
 - [293] K.M. Kahn. Uniform : a language based upon unification. *IJCAI*, 7 :933–939, 1981.
 - [294] R. Karp. *On the computational complexity of combinatorial problems*. Networks, 1975.
 - [295] D. Kayser. Examen de diverses méthodes utilisées en représentation des connaissances. In *Examen de diverses méthodes utilisées en représentation des connaissances*. Congrès d'Intelligence Artificielle et Reconnaissance des formes, AFCET, 1984.
 - [296] D. Kayser and D. Coulon. Variable-depth natural language understanding. *IJCAI*, 7 :64–66, 1981.
 - [297] D. Kliber and P. Morris. Don't be stupid. *IJCAI*, 7 :345–347, 1981.
 - [298] G. Kling. *A paradigm for reasoning by analogy*. A.I., 1971.
 - [299] F. Klix. *Human and artificial intelligence*. North Holland, 1979.
 - [300] D. Knuth and R. Moore. *An analysis of Alpha-Beta pruning*. A.I., 1975.
 - [301] Y. Kodratoff, J. G. Ganascia, B. Clavieras, T. Bollinger, and G. Tecuci. *Careful generalization for concept learning*. ECAI, 1984.
 - [302] Y. Kodratoff and J. Sailantin. *Outils pour l'apprentissage*. GR 22, 1983.
 - [303] K. Konolige. An inference net compiler for the prospector rule-based consultation system. *IJCAI*, 6 :487–489, 1979.
 - [304] B. Kordiemskeii. *Sur le sentier des mathématiques*. Dunod, 1963.
 - [305] R. E. Korf. Toward a model of representation changes. *A.I.*, 14, 1980.
 - [306] R. Kowalski. *Search startegies for theorem proving*. M.I., 1970.
 - [307] R. Kowalski. *Linear resolution with selector function*. A.I., 1971.
 - [308] R. Kowalski. Predicate logic as a programming language. *Proc. IFIP*, 3 :569–574, 1974.
 - [309] R. Kowalski. *Logic for Problem Solving*. North-Holland, New York, 1979.
 - [310] R. Kowalski. *Logic for problem solving*. North Holland, 1980.
 - [311] M. Kraitchich. *La mathématique des jeux*. Dunod.
 - [312] C.A. Kullkowsky and S. Weiss. Representation of expert knowledge for consultation : the casnet and expert projects. *Artificial Intelligence in Medicine*, 1982.
 - [313] C.A. Kullkowsky, S. Weiss, M. Trigoboff, and A. Safir. Clinical consultation and the representation of disease process : some a.i. approaches. Technical Report Report CEM-TR 58, Rutgers University, 1976.

- [314] A. Rich L. Siklossy and V. Marinov. *Breadth-first search : some surprising results.* A.I., 1973.
- [315] Dreussi L. Siklossy. *An efficient robot planner which generates its own procedures.* IJCAI, 1973.
- [316] D. G. Carlson L. T. Wos and G. A. Robins. *Efficiency and completeness of the set of support strategy in theorem-proving.* ACM, 1965.
- [317] M.S. Lagrange and M. Renaud. Deux expériences de simulation du raisonnement en archéologie au moyen d'un système expert : le système snark. *Informatique et sciences humaines*, pages 59–60, 161–188, 1984.
- [318] M.S. Lagrange and M. Renaud. Intelligent knowledge-based systems in archaeology : a computerized simulation of reasoning by means of an expert system. *Computer and Humanities*, 19 :37–52, 1985.
- [319] P. Langley. Rediscovering physics with bacon-3. *IJCAI*, 6 :161–188, 1979.
- [320] P. W. Langley. *Rediscovering physics with BACON 3.* IJCAI, 1977.
- [321] P. W. Langley. *Data-driven discovery.* C.S., 1981.
- [322] C. Lasserre. *Rapport de la logique mathématique dans les systèmes de décision en robotique.* PhD thesis, Toulouse, 1978.
- [323] J. C. Latombe. *AI and pattern recognition in CAD.* North Holland, 1978.
- [324] J. C. Latombe. Intelligence artificielle et robotique industrielle. *Le nouvel Automatisme*, 1979.
- [325] J.C. Latombe. *Une application de l'intelligence artificielle à la conception assistée par ordinateur.* PhD thesis, Grenoble, 1977.
- [326] J. P. Laurent. *Un programme qui calcule les limites en levant les indéterminations par des procédés heuristiques.* Université Paris VI, 1972.
- [327] J. P. Laurent. *A program that computes limits using heuristics to evaluate the indeterminate forms.* A.I., 1973.
- [328] J. P. Laurent. *Un système qui met en évidence des erreurs sémantiques dans les programmes.* Université Paris VI, 1978.
- [329] J. L. Laurière. A language and a program for stating and solving combinatorial problems. *A.I.*, 10, 1978.
- [330] J. L. Laurière. A language and a program for stating and solving combinatorial problems. *A.I.*, 10, 1978.
- [331] J. L. Laurière. *Eléments de programmation dynamique.* Gauthier-Villars, 1979.
- [332] J. L. Laurière. *Eléments de programmation dynamique.* Gauthier-Villars, 1979.
- [333] J.L. Laurière. *Sur la coloration de certains hypergraphes. Application aux problèmes d'emploi du temps.* PhD thesis, Paris VI, 1971.
- [334] J.L. Laurière. Problèmes d'emploi du temps (e, h, r). *Actes des journées combinatoires Af cet*, 1972.
- [335] J.L. Laurière. Problèmes d'emploi du temps et algorithme de coloration des hypergraphes. *C.R. Acad. Sci Paris 279 a.*, pages 1159–1162, 1974.
- [336] J.L. Laurière. Un programme d'élaboration des emplois du temps scolaires. *Congrès Af cet aide à la décision*, 1974.
- [337] J.L. Lauriere. Représentation et utilisation des connaissances. *Publications du GR 22. Programmes d'intelligence artificielle utilisant une grande quantité de connaissances*, pages 3–112, 1979.

-
- [338] J.L. Lauriere. Représentation et utilisation des connaissances. *Publications du GR 22. Programmes d'intelligence artificielle utilisant une grande quantité de connaissances*, pages 3–112, 1979.
 - [339] J.L. Lauriere. Représentation et utilisation des connaissances. deuxième partie : représentation des connaissances. *Technique et science informatique*, 2 :25–42, 1982.
 - [340] J.L. Lauriere. Représentation et utilisation des connaissances. première partie : les systèmes experts. *Technique et science informatique*, 1 :25–42, 1982.
 - [341] J.L. Laurière. *Un langage et un programme pour énoncer et résoudre des problèmes combinatoires*. PhD thesis, Paris VI, 1982.
 - [342] J.L. Laurière. Représentation et utilisation de connaissances. *TSI n° 1 et 2*, 1983.
 - [343] J.L. Laurière. Un langage déclaratif snark. *Revue TSI*, 3, 1986.
 - [344] J.L. Laurière and M. Vialatte. Manuel d'utilisation du moteur d'inférence snark. Technical report, Institut de programmation, 1985.
 - [345] E. Lawier. *A note on the complexity of the chromatic nimber problem*. Information Processing Letter, 1976.
 - [346] E. Lawier and D. Wood. *Branch and bound methods : a survey*. Opn. res., 1966.
 - [347] R. Le Faivre. Fuzzy problem solving. Technical Report 37, University of Wisconsin, 1974.
 - [348] W. Lehnert, M. G. Dyer, P. N. Johnson, C. J. Yang, and S. Harley. Boris an experiment in in-depth understanding of narratives. *A.I.*, 20, 1983.
 - [349] C. Lemaître. *Problèmes de planification et apprentissage dans le cas d'un programme de simulation de robot*. Université Paris VI, 1974.
 - [350] D. Lenat. Beings : knowledge as interacting expert. *IJCAI*, 4, 1975.
 - [351] D. Lenat. ubiquity of discovery. *A.I.*, 9 (3) :257–286, 1977.
 - [352] D. Lenat and J. McDermott. Less than general production systems architecture. *IJCAI*, 5 :928–932, 1977.
 - [353] D. B. Lenat. *The ubiquity of discovery*. A.I., 1977.
 - [354] D. B. Lenat. *On automated scientific theory formation : case study using the AM program*. Davis & Lenat eds, 1980.
 - [355] D. B. Lenat. *The nature of heuristics*. A.I., 1982.
 - [356] C. Lesveque. *Un système expert en paye et gestion du personnel*. PhD thesis, Paris VI, 1984.
 - [357] H.J. Levesque. *The logic of incomplete knowledge bases in Conceptual Modelling : perspectives from artificial intelligence*. Databases and programming languages. Springer-Verlag, 1983.
 - [358] J. Lighthill. *Artificial Intelligence : A general survey*. Science Research Council, 1973.
 - [359] P. H. Lindsay and D. A. Norman. *Traitement de l'information et comportement humain*. Etudes Vivantes, 1977.
 - [360] R.K. Lindsay, B.G. Buchanan, E.A. Feigenbaum, and J. Lederberg. *Applications of artificial intelligence for organic chemistry : The Dendral projects*. McGraw-Hill, 1980.
 - [361] M. Lopez. Réalisation d'une interface de communication en langue naturelle avec le système tropic. In *Réalisation d'une interface de communication en langue naturelle avec le système TROPIC*. AFCET, 1979.

- [362] E. Lucas. *Récréations mathématiques*. A. Blanchard, Paris, 1960.
- [363] B. Fade M. Cayrol and H. Farreny. *Formal objets and feature associations in ARGOS-II*. IJCAI, 1979.
- [364] B. Fade M. Cayrol and H. Farreny. *Objets formels et attributs dans ARGOS-II*. Congrès AFCET, Toulouse, 1979.
- [365] B. Fade M. Cayrol and H. Farreny. *ARGOS-II un système de production pour écrire des résolveurs experts*. Actes du Congrès AFCET Informatique, Nancy, 1980.
- [366] A. K. Mackworth. Consistency in networks of relations. *A.I.*, 1977.
- [367] A. K. Mackworth. Consistency in networks of relations. *A.I.*, 1977.
- [368] Z. Manna and R. Waldinger. Knowledge and reasoning in program synthesis. *A.I.*, 6 :175–208, 1975.
- [369] M. H. Marcus. *A theory of syntactic recognition for natural language*. PhD thesis, MIT, 1980.
- [370] A. Markov. *The theory of algorithms*. US Dept. of Commerce, translated from Russian by J. Shorr-Kon, 1954.
- [371] D. Marr. *Early processing of visual information*. B. Philosophical Transactions of the Royal Society of London, 1976.
- [372] D. Marr. Artificial intelligence - a personal view. *A.I.*, 1977.
- [373] Mathiab-Group. *The Macsyma reference manual*. MIT lab. for computer science, 1974.
- [374] J. Mathieu and R. Thomas. *Manuel de psychologie*. Vigot, 1985.
- [375] McCarthy. Epistemological problems in ai. *IJCAI*, 5 :1036–1044, 1977.
- [376] J. McCarthy. Programs with common sense. *Semantic information processing*, pages 403–409, 1968.
- [377] J. McCarthy. Circumscription - a form of non-monotonic reasoning. *A.I.*, 13 :27–39, 1980.
- [378] D. McCracken. Representation and efficiency in a production system for speech understanding. *IJCAI*, 6 :556, 1979.
- [379] D. McDermott. *A deductive model of control of a problem solver*. SIGART, 1977.
- [380] D. McDermott. *Production system conflict resolution strategies*, pages 177–201. 1977.
- [381] D. McDermott. Planning and acting. *C.S.*, 2 :71–109, 1978.
- [382] D. McDermott. Learning to use analogies. *IJCAI*, 6 :568–576, 1979.
- [383] D. McDermott. Extending a knowledge-based system to deal with ad-hoc constraints. *IJCAI*, 7 :824–828, 1981.
- [384] D. McDermott. Ri : a rule-based configurer of computer systems. *A.I.*, 18 :39–88, 1982.
- [385] D. McDermott and J. Doyle. Non-monotonic logic ii : Non-monotonic modal theories. Technical Report 174, Computer Science Dept., Yale University, 1980.
- [386] D. McDermott, A. Newell, and J. Moore. *The efficiency of certain production system implementations*, pages 155–176. 1977.
- [387] D. D. McDonald. *Language production as a process of decision making under constraints*. PhD thesis, MIT, 1980.
- [388] W. Van Melle. A domain independent production rule system for consultation systems. *IJCAI*, 6, 1979.

-
- [389] H. Meloni. *Etude et réalisation d'un système de reconnaissance automatique de la parole continue*. PhD thesis, Université d'Aix-Marseille 2, Luminy, 1982.
- [390] B. Meltzer and D. Michie. *Machine Intelligence, Vol. 4*. Elsevier, 1969.
- [391] B. Meltzer and D. Michie. *Machine Intelligence, Vol. 5*. Elsevier, 1970.
- [392] B. Meltzer and D. Michie. *Machine Intelligence, Vol. 6*. Elsevier, 1971.
- [393] B. Meltzer and D. Michie. *Machine Intelligence, Vol. 7*. Elsevier, 1972.
- [394] B. Merialdo. *Représentation des ensembles en démonstration automatique de théorèmes*. PhD thesis, Université Paris VI, 1979.
- [395] A. Meyer. Ten thousand and one logics of programming. *ILP, MIT 6380*, 1980.
- [396] B. Meyer and C. Baudoin. *Méthodes de programmation*. Eyrolles, 1978.
- [397] R. S. Michalski. *A theory and methodology of inductive learning*. A.I., 1983.
- [398] R. S. Michalski and R. L. Chilausky. *Learning by being told and learning from examples : An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnostic*. International Journal of Policy Analysis and Information Systems, 1980.
- [399] Michie. *Expert Systems in the Microelectronic Age*. Edinburgh University Press, 1980.
- [400] D. Michie. *Machine Intelligence, Vol. 3*. Elsevier, 1968.
- [401] G. A. Miller. The magical number seven, plus or minus two : Some limits of our capacity for processing information. *Psychological Review*, 1956.
- [402] M. Minsky. *Semantic Information processing*. Cambridge, Mass. MIT Press., 1968.
- [403] M. Minsky. *A framework for representing knowledge*, chapter In Winston 75, pages 211–277. 1975.
- [404] M. L. Minsky. *Matter, minds and models*. IFIP, 1965.
- [405] M. L. Minsky and S. Papert. *Perceptrons : an introduction to computational geometry*. PhD thesis, MIT, 1969.
- [406] T. M. Mitchell. *An analysis of generalization as a search problem*. IJCAI, 1979.
- [407] T. M. Mitchell. *Learning and problem solving*. IJCAI, 1983.
- [408] T.M. Mitchell, P. Utgoff, B. Nudel, and R. Banerji. Learning problem solving heuristics through practice. *IJCAI*, 7 :127–134.
- [409] T. Moran. The symbolic nature of visual imagery. *IJCAI*, 3 :472–477, 1973.
- [410] J. Moses. *Symbolic integration*. PhD thesis, MIT, 1967.
- [411] J. Moses. *Algebraic simplification : A guide for the perplexed*. ACM, 1971.
- [412] J. D. Myers and H. Pople. Internist a consultative diagnostic program. In *Internal medicine proceeding of the 1st annual symposium on computer applications in medical care*, 1977.
- [413] J. Mylopoulos. An overview of knowledge representation. *SIGPLAN*, 16 (1) :5–12, 1981.
- [414] J. Von Neuman. *Theory of games and economic behavior*. PhD thesis, Princeton University, 1944.
- [415] A. J. Nevins. *A human oriented logic for automatic theorem proving*. ACM, 1974.
- [416] A. J. Nevins. Plane geometry theorem proving using forward chaining. *A.I.*, 6, 1975.

- [417] A. J. Nevins. *A relaxation approach to splitting in an automatic theorem prover.* A.I., 1975.
- [418] A. Newell. *Artificial Intelligence and the Concept of Mind.* Roger Schank and Kenneth Colby, 1973.
- [419] A. Newell. *Visual Information Processing*, chapter Production systems : models of control structures, pages 463–526. 1976.
- [420] A. Newell. The knowledge level. *A.I.*, 1982.
- [421] A. Newell. The knowledge level. *A.I.*, 18 (1) :87–128, 1982.
- [422] A. Newell and J. McDermott. Psg manual. Technical report, Carnegie and Mellon University, 1975.
- [423] A. Newell, J. C. Shaw, and H. A. Simon. *Programming the logic theory machine.* Western joint computer conference, 1957.
- [424] A. Newell and H. Simon. *Human problem solving.* Prentice Hall, 1972.
- [425] A. Newell and H. Simon. *Human problem solving.* Prentice Hall, 1982.
- [426] A. Newell and H. A. Simon. *The logic theory machine : a complex information processing system.* IRE Trans on Information Theory, IT-2, 1956.
- [427] A. J. Newins. *Plane geometry theorem proving using forward chaining.* A. I., 1975.
- [428] N. Nil and N. Allelo. Age : attempt to generalize. Technical report, Stanford, 1978.
- [429] N. Nil and E. Feigenbaum. *Rule based understanding of signals.* Waterman and Hayes-Roth (eds), 1978.
- [430] N. J. Nilsson. *Problem solving methods in A.I.* McGraw-Hill Book Company, 1971.
- [431] N. J. Nilsson. *Principes of A.I.* Tioga Pub. Co., 1980.
- [432] J. Nivergelt, J. Craig Farrar, and E. M. Reingold. *Computer approaches to mathematical problems.* Prentice Hall, 1974.
- [433] D. A. Norman, D. E. Rumelhart, and the LNR Research Group. *Explorations in cognition.* Freeman, 1975.
- [434] A. Papert. *Teaching children thinking.* Programmed Learning and Educational Technology, 1972.
- [435] S. Papert. *Mindstorms, children, Computers and powerful ideas.* Basic Books, 1980.
- [436] D. Pastre. *Démonstration automatique de théorèmes en théorie des ensembles.* Université Paris VI, 1976.
- [437] D. Pastre. Automatic theorem proving in set theory. *A.I.*, 10, 1978.
- [438] D. Pastre. Automatic theorem proving in set theory. *A.I.*, 10 :1–27, 1978.
- [439] D. Pastre. Automatic theorem proving in set theory. *A.I.*, 10, 1978.
- [440] D. Pastre. *Observation du mathématicien : aide à l'enseignement et à la démonstration automatique de théorèmes*, volume 9. Educational studies in mathematics, 1978.
- [441] D. Pastre. *Muscadet, un système de démonstration automatique de théorèmes utilisant des connaissances et des métacatégories en mathématiques.* Université Paris VI, 1984.
- [442] R. Patil, P. Szolovits, and W. B. Schwartz. Modeling knowledge of the patient in acid-base and electrolyte disorders. *Artificial Intelligence in Medicine*, 1982.

-
- [443] C. A. Petri. *Communication with automata*. PhD thesis, Bonn., 1962.
- [444] C. E. Pfefferkorn. *A heuristic problem solving design system for equipment layout*. C. ACM, 1975.
- [445] J. M. Pierrel. Utilisation de contraintes linguistiques en compréhension automatique de la parole continue : le système myrtille-ii. *TSI*, 1(5) :403–421, 1982.
- [446] S. Pinson. Représentation des connaissances dans les systèmes experts. *RAIRO Informatique*, 15(4) :343–367, 1980.
- [447] J. Pitrat. *Utilisation de connaissances déclaratives*. Ecole d’été AFCET, Aix en Provence.
- [448] J. Pitrat. *Un programme de démonstration de théorèmes*, volume 7. Monographie AFCET, Dunod, 1970.
- [449] J. Pitrat. *Un programme de démonstration de théorèmes*. Dunod, 1970.
- [450] J. Pitrat. *A general game playing program*. Edinburgh University Press, 1971.
- [451] J. Pitrat. *A program for learning to play chess*. Chen Academic Press, 1976.
- [452] J. Pitrat. *A chess combination program which uses plans*. A.I., 1977.
- [453] J. Pitrat. *Formalism for text analysis*. Bonas, 1977.
- [454] J. Pitrat. *Realization of a program learning to find combinations at chess*. Computed oriented learning processes, 1978.
- [455] J. Pitrat. *Réalisation d’un analyseur générateur lexicographique général*. PhD thesis, Université Paris VI, 1979.
- [456] J. Pitrat. Un interpréteur de contraintes pour la compréhension du langage naturel. In *Un interpréteur de contraintes pour la compréhension du langage naturel*. Colloque d’intelligence artificielle, Caen, 1980.
- [457] J. Pitrat. *Textes, ordinateurs et compréhension*. Eyrolles, 1985.
- [458] D. A. Plaisted. *A simplified problem reduction format*. A.I., 1982.
- [459] I. Pohl. *Heuristic search viewed as path finding in a graph*. A.I., 1970.
- [460] I. Pohl. Heuristic search viewed as path finding in a graph. A.I., NR.
- [461] G. Polya. *Induction and analogy in mathematics*. Princeton et Gauthier-Villars, 1954.
- [462] G. Polya. *Mathematics and plausible reasoning*. PhD thesis, Princeton University, 1954.
- [463] G. Polya. *Mathematical discovery*. Double Day, 1957.
- [464] G. Polya. *How to solve it*. Princeton University Press and Dunod, 1959.
- [465] G. Polya. *Mathematical discovery*. Wiley, 1962.
- [466] G. Polya. *Comment poser et résoudre un problème*. Dunod, 1965.
- [467] R. Popesco. *CHELEM, un système-expert pour trouver la ligne de jeu du déclarant au bridge*. Université Paris VI, 1984.
- [468] E. Post. Finite combination processes. *Journal of symbolic logic*, 1 :103–105, 1936.
- [469] E. Post. Formal reductions of the general combinatorial decision problem. *American journal of mathematics*, (65) :197–215, 1943.
- [470] H. Prade. A synthetic view of approximate reasoning techniques. *IJCAI*, 8 :130–136, 1983.
- [471] L. Pun. *Systèmes industriels d’intelligence artificielle*. Editions Tests, 1984.
- [472] C. Queinnec. *Langage d’un autre type : LISP*. Eyrolles, 1980.

- [473] R. M. Quillian. *The teachable language comprehender : a simulation program and theory of language*, volume 18. CACM, 1969.
- [474] J. Quinian. Knowledge-based system for locating missing high cards in bridge. *IJCAI*, 6 :705–707.
- [475] P. E. Hart R. E. Fikes and N. J. Nilsson. *Learning and executing generalized robot plans*. A.I., 1972.
- [476] P. E. Hart R. E. Fikes and N. J. Nilsson. *Some new directions in robot problem solving*. M.I., 1972.
- [477] J. G. Caronnel R. S. Michalski and T. M. Mitcheli. *Machine learning, an artificial intelligence approach*. Tioga Pub Co., 1983.
- [478] B. Raphaël. *The thinking computer*. Freeman, 1976.
- [479] R. Reiter. On reasoning by default. *Theoretical issues in Natural Language Processing-2*, pages 210–218, 1978.
- [480] R. Reiter. A logic for default reasoning. *A.I.*, 13 (1-2) :81–132, 1980.
- [481] R. Reiter. A logic for default reasoning. *A.I.*, 13 (1-2) :81–132, 1980.
- [482] E. Rich. *Artificial Intelligence*. McGraw-Hill Book Company, 1983.
- [483] C. Rieger. An organization of knowledge for problem solving and language comprehension. *A.I.*, 7 :89–127, 1976.
- [484] G. A. Robinson. *The generalized resolution principle*. M.I., 1973.
- [485] G. A. Robinson and L. Wos. *Paramodulation and theorem-proving in first order theories with equality*. M.I., 1969.
- [486] J. A. Robinson. *A machine-oriented logic based on the resolution principle*. ACM, 1965.
- [487] G. A. Robison. *Mechanizing higher-order logic*. M.I., 1974.
- [488] P. S. Rosembloom. *A world-championship-level othello program*. A.I., 1982.
- [489] S. Rosenberg. Hprl : a language for building expert systems. *IJCAI*, pages 215–217, 1983.
- [490] M. Rousseau. *Résolution automatique d'exercices d'électricité posés en français*. PhD thesis, Paris VI, 1973.
- [491] P. Roussel. *Prolog : manuel de référence et d'utilisation*. Groupe d'intelligence artificielle, Marseille Luminy, 1975.
- [492] F. Rousselot. *Simulation d'un robot qui comprend et exécute des ordres donnés en français*. Université Paris VI, 1975.
- [493] M. C. Rousset. *TANGO, moteur d'inférences avec variables*. PhD thesis, Orsay, 1983.
- [494] M. D. Rychener. *Production systems as a programming language for artificial intelligence Applications*. PhD thesis, Carnegie Mellon, 1976.
- [495] M.D. Rychener. *Production systems as a programming language for artificial intelligence Applications*. PhD thesis, Carnegie Mellon, 1976.
- [496] G. Sabah. *Contribution à la compréhension effective d'un récit*. PhD thesis, Université Paris VI, 1980.
- [497] G. Sabah and M. Rady. A deterministic syntactic semantic parser. *IJCAI*, 8, 1983.
- [498] E. D. Sacerdoti. *Planning in a hierarchy of abstraction spaces*. A.I., 1974.
- [499] E. D. Sacerdoti. *The nonlinear nature of plans*. IJCAI, 1975.

-
- [500] E. D. Sacerdoti. *A Structure for plans and behavior*. Elsevier, 1977.
 - [501] M. Saikoff. *Une grammaire en chaîne du français*. Dunod, 1973.
 - [502] J. Sallentin. *Représentation d'observations dans le contexte de la théorie de l'information*. Université Paris VI, 1979.
 - [503] J. Sallentin and J. Quinqueton. *Algorithms for learning logical formulas*. IJCAI, 1983.
 - [504] A. L. Samuel. *Some studies in machine learning using the game of checkers*. IBM J. Research and Development, 1959.
 - [505] A. L. Samuel. *Some studies in machine learning using the game of checkers II*. IBM J. Research end Development, 1967.
 - [506] R. Schank and R. Abelson. *Scripts, plans, goals, and understanding*. Lawrence Eribaum associates, 1977.
 - [507] R. C. Schank. *Conceptual information processing*. North Holland, 1975.
 - [508] R.C. Schank. Conceptual dependency : a theory of natural language understanding. *Cognitive Psychology*, 3 :552–631, 1972.
 - [509] R.C. Schank. *Conceptual Information Processing*. Elsevier Science Ltd, 1975.
 - [510] R.C. Schank and R.P. Abelson. *Scripts, plans, goals, and understanding*. Hillsdale, N.J., Lawrence Eribaum, 1977.
 - [511] C. E. Shannon. Programming a computer to play chess, 1950.
 - [512] S. C. Shapiro. *Techniques o fAI*. Van Nostrand, 1979.
 - [513] E. Shortliffe. *Computer-based medical consultations : MYCIN*. Elsevier, 1976.
 - [514] E. Shortliffe, A. Scott Carlisle, M. B. Bischoft, A. B. Campbell, W. Van Melle, and D. Jacobs. Oncocin : an expert system for oncology protocol management. *IJCAI*, 7 :876–981, 1981.
 - [515] E.H. Shortliffe and B.G. Buchanan. A model of inexact reasoning in medicine. *Mathematical Bioscience*, 23 :351–379, 1975.
 - [516] L. Siklossy and Marinov. *Heuristic search versus exhaustive search*. IJCAI, 1971.
 - [517] L. Siklossy and J. Roach. *Model verification and improvement using DISPRO-VER*. A.I., 1975.
 - [518] L. Siklossy. *Let's talk LISP*. Prentice Hall, 1976.
 - [519] R. F. Simmons. *Semantic networks : their computation and use for understanding english sentences*. in Schank and Colby 1973, 1973.
 - [520] H. Simon. *Why should machines learn*. Carbonell & Mitcheli eds, Tioga Pub Co., 1983.
 - [521] H. A. Simon. *Sciences of the artificial*. PhD thesis, MIT, 1969.
 - [522] H. A. Simon. The structure of ill structured problems. *A.I.*, 1973.
 - [523] H. A. Simon. *Models of thought*. PhD thesis, Yale University, 1979.
 - [524] H. A. Simon and Barenfeld. Information processing analysis of perceptual processes in problem solving. *Psychological Review*, 1969.
 - [525] H. A. Simon and L. Siklossy. *Representation and meaning*. Prentice Hall, 1973.
 - [526] G. L. Simons. *Introducy Artificial intelligence*. NCC Pub, 1984.
 - [527] T. Skolem. *From Frege to Gödel*, chapter The foundations of elementary arithmetic establish by means of the recursive model of thought, without the use of apparent variables ranging over infinite domains. Harvard University Press, 1967.

- [528] J. R. Slagle. *A heuristic program that solves symbolic integration problems in freshman calculus.* McGraw-Hill Book Company, 1963.
- [529] J. R. Slagle. *A.I. : the heuristic programming approach.* McGraw-Hill Book Company, 1971.
- [530] A.I. Software. *The international directory of A.I. Companies.* SRL. Rovigo Italie, 1984.
- [531] G. L. Steele. *Common LISP.* Digital Press, 1984.
- [532] M. Stefik. Inferring dna structures from segmentation data. *A.I.*, 11(1) :85–114, 1978.
- [533] M. Stefik. An examination of a frame-structured representation system. *IJCAI*, 6 :845–852, 1979.
- [534] M. Stefik. Planning and meta-planning (molgen 2). *A.I.*, 16(2) :141–170, 1981.
- [535] M. Stefik. Planning with constraints (molgen 1). *A.I.*, 16(2) :111–140, 1981.
- [536] M. Stefik, J. Atkins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti. The organization of expert systems, a tutorial. *A.I.*, 18 (2) :135–173, 1982.
- [537] G. Sussman and R. Stallman. Heuristic techniques in computer-aided circuit analysis. *CAS*, 22 :857–865, 1975.
- [538] G. J. Sussman. *A computer model of skill acquisition*, volume 1. Elsevier, 1975.
- [539] G. J. Sussman and G. L. Steele. *CONSTRAINTS - a language for expressing almost-hierarchical descriptions.* A.I., 1980.
- [540] W. R. Swartout. Explaining and justifying expert consulting programs. *IJCAI*, 7 :815–823, 1979.
- [541] P. Szeredi. Prolog, a very high language based on predicate logic. *Hungarian Conference on computer science*, 1977.
- [542] P. Szolovits and S.G. Pauker. Categorical and probabilistic reasoning in medical diagnosis. *A.I.*, 11 :115–144, 1978.
- [543] Y. Tabourier. Problèmes de normalisation et de décomposition dans les modèles conceptuels de données. *Informatique et gestion*, n° 133 :72–81, 1982.
- [544] R. Tarjan. *Depth first search and linear graph algorithms.* Siam. J. Comput., 1972.
- [545] F. Tonge. *A heuristic program for assembly line balancing.* Prentice Hall, 1961.
- [546] F. Tonge. *A heuristic program for assembly line balancing.* Prentice Hall, 1961.
- [547] F. Tonge. *A heuristic programm for assembly line balancing.* Prentice Hall, 1961.
- [548] F. Tonge. *A heuristic programm for assembly line balancing.* Prentice Hall, 1961.
- [549] J. Tricot. *L'ordinateur et les jeux.* L'informatique nouvelle, 1975.
- [550] A. M. Turing. *Computational machinery and intelligence.* Mind, 1950.
- [551] L. Uhr. Parallel-serial productions systems. *IJCAI*, 6 :911–916, 1979.
- [552] W. Van Melle. A domain-independent system that aids in constructing knowledge-based consultation programs. Technical Report 820, Computer Science Dept., Stanford University, 1980.
- [553] R. Venken. *A Prolog Meta-interpreter for partial evaluation and its application to source transformation and query-optimisation.* ECAI, 1984.
- [554] M. F. Verdejo. *Etude du langage naturel. Simulation d'un robot capable de mener un dialogue en espagnol.* PhD thesis, Université Paris VI, 1975.

-
- [555] S. A. Vere. *Induction of concepts in the predicate calculus*. IJCAI, 1975.
 - [556] M. Vialatte. *Introduction de métaconnaissances, de gestion d'hypothèses, de logique d'ordre 0 et 2 dans SNARK*. Institut de programmation, 1984.
 - [557] M. Vialatte. *Description et application du moteur d'inférence SNARK*. PhD thesis, Université Paris VI, 1985.
 - [558] F. Vidal. *Problem solving*. Dunod, 1971.
 - [559] A. Vilnat and G. Sabah. How a system may be self-conscious. ECAI, 1984.
 - [560] M. Vivet. *Un programme qui vérifie des identités en utilisant le raisonnement par récurrence*. Université Paris VI, 1973.
 - [561] M. Vivet. *CAMELIA : A knowledge based mathematical system*. ECAI, 1984.
 - [562] R.S. Boyer W. W. Bledsoe and W. H. Henneman. *Computer proofs of limit theorems*. A.I., 1972.
 - [563] D. L. Waltz. *An english language question answering system for a large relational database*. McGraw-Hill Book Company, 1981.
 - [564] D. H. Warren. Implementing prolog : D.a.i. Technical report, University of Edinburgh, 1977.
 - [565] D. Waterman. Adaptative production systems. IJCAI, 4 :296–303, 1975.
 - [566] D. Waterman. Generalization learning techniques for automating the learning of heuristics. A.I., 1 :121–170, 1975.
 - [567] D. Waterman and A. Newell. Protocol analysis as a task for artificial intelligence. A.I., 2 :285–318, 1971.
 - [568] D. Waterman and F. Hayes Roth. *Pattern directed inference systems*. Academic Press, New York, 1978.
 - [569] D. Waterman and F. Hayes Roth. *Pattern directed inference systems*. Academic Press, New York, 1978.
 - [570] M. N. Wegman. *Linear unification*. J. of Comp. and System Sciences, 1978.
 - [571] S. M. Weiss and C. A. Kullkowsky. Expert : a system for developing consultation models. IJCAI, 6 :942–947, 1978.
 - [572] J. Weizenbaum. *Eliza, a computer program for the study of natural language communication between man and machine*. CACM, 1966.
 - [573] D. J. Welsh. *An upper bound for the chromatic number of a graph and its application to the time-tabling problem*. Computer J., 1968.
 - [574] H. Wertz. *Intelligence Artificielle*. Masson, 1984.
 - [575] H. Wertz. *LISP. Une introduction à la programmation*. Masson, 1985.
 - [576] R.W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. A.I., 13 (1-2) :133–170, 1980.
 - [577] W. Wickelgren. *How to solve problems*. Freeman, 1974.
 - [578] R. Wilensky. Meta planning. C.S., 5 :197–233, 1981.
 - [579] D. Wilkins. Using plans in chess. IJCAI, 6 :960–967, 1979.
 - [580] D. Wilkins. *Using plans in chess*. IJCAI, 1979.
 - [581] D. Wilkins. *Using patterns and plans in chess*. A.I., 1980.
 - [582] D. Wilkins. *Using knowledge to control tree chess*. A.I., 1982.
 - [583] Y. Wilks. *An intelligent analyzer and understander of English*. CACM, 1975.
 - [584] R. Willensky. Understanding goal-based stories. Technical Report 140, Yale University, 1978.

- [585] Winograd. *Representation and Understanding*, chapter Frame representations and the declarative/procedural controversy. Academic Press, 1975.
- [586] T. Winograd. *Understanding natural language*. Edinburgh University Press, 1972.
- [587] T. Winograd. *Frame representations and the declarative procedural controversy*. Academic Press, 1975.
- [588] T. Winograd. *What does it mean to understand language?*, volume 4. C. S., 1980.
- [589] T. Winograd. Des machines savantes, mais incultes. *La science des robots, Science et Vie hors-série*, pages 76–83, 1982.
- [590] T. Winograd. *Language as a cognitive process*. Addison-Wesley, 1983.
- [591] H. Winston and R. H. Brown. *Artificial Intelligence. An MIT perspective*. PhD thesis, 1979.
- [592] P. H. Winston. *Learning structural descriptions from examples*. M.I.T., 1970.
- [593] P. H. Winston. *Learning structural descriptions from examples*. McGraw-Hill Book Company, 1975.
- [594] P. H. Winston. *The psychology of computer vision*. McGraw-Hill Book Company, 1975.
- [595] P. H. Winston. *The psychology of computer vision*. Mc Graw-Hill, 1975.
- [596] P. H. Winston. *Artificial Intelligence*. Addison-Wesley, 1977.
- [597] P. H. Winston. *Artificial Intelligence*. Addison Welsey, 1977.
- [598] P. H. Winston. *Learning by creating and justifying frames*. A.I., 1978.
- [599] W. Woods. *Representation and Understanding*, chapter What's in a link ? Foundations for semantic networks, pages 35–82. Academic Press, 1975.
- [600] W. Woods. *Representation and Understanding*, chapter What's in a link ? Foundations for semantic networks, pages 35–82. Academic Press, 1975.
- [601] W. A. Woods. *Transition networks grammars for natural language analysis*, volume 13. CACM, 1970.
- [602] H. Wooldridge. *The machine of the brain*. McGraw-Hill Book Company, 1963.
- [603] R. Woolsey. *How to do integer programming in the real world in integer programming*. Addison Welsey, 1965.
- [604] L.A. Zadey. Approximate reasoning based on fuzzy logic. *IJCAI*, 6 :1004–1010, 1979.
- [605] L.A. et al. Zadey. *Fuzzy sets and their applications to cognitive and decision processes*. Academic Press, 1975.
- [606] G. P. Zarri. Building the inference component of an historical information retrieval system. *IJCAI*, 7 :401–408, 1981.