IA-Résolution de problèmes
2ème année SRI

M.C. Lagasquie

4 février 2021

Table des matières

Chapitre 1

Planning

Séance	Durée	Contenu
1	(2h)	Intro : qu'est-ce que l'IA ? Qu'est ce que la résolution de problème ? Les différents types de problèmes
2	(2h)	Logique des propositions : vocabulaire, syntaxe et sémantique, théorie des modèles, problème SAT
3-4	(4h)	Complexité des algorithmes, machine de Turing, complexité des problèmes
5	(2h)	Représentation de problème en logique
6	(2h)	Espace d'état : définition et utilisation sur divers exemples
7	(2h)	Méthodes complètes non informées (profondeur, largeur) ou informées (Dijkstra, A*)
8	(2h)	Méthodes approchées (hill climbing, tabou)
9-10	(4h)	Formalisme et résolution des CSP (backtrack sans et avec ordonnancement)
11	(2h)	Programmation linéaire par nombres entiers (PLNE) : les idées et la méthode, des exemples

Chapitre 2

Introduction

2.1 Contexte

Ce cours est destiné à donner qqs clés de compréhension des mécanismes de base mis en œuvre et étudiés en ***IA (Intelligence Artificielle)*** dès lors qu'on cherche à faire de la ***résolution de problèmes***.

IA – Intelligence Artificielle Il s'agit de la discipline destinée à simuler un comportement "intelligent", à obtenir une machine avec un comportement rationnel (pouvant prendre en compte son environnement et interagir avec lui de manière rationnelle).
Les domaines d'utilisation sont ceux que l'humain ne peut traiter seul :
- problèmes combinatoires (trop de possibilités à gérer pour pouvoir le faire au niveau d'un être humain),
- problèmes stockastiques/aléatoires (impossibilité de prédire à l'avance ce qui va se passer),
- situations de concurrence (gestion des conflits).

Résolution de problèmes Pour résoudre un problème, il faut suivre une méthodologie assurant qu'une solution pourra être trouvée *dans tous les cas*.
La première étape est la représentation formelle (formulation, codage) du problème. Cette étape nécessite de bien poser le problème, à un certain niveau d'abstraction.
La seconde étape sera alors de résoudre le problème avec les outils adaptés à la formulation choisie.

Plan du cours Nous allons tout d'abord décrire les différents types de problèmes auxquels on peut être confronté.
Cela nous amènera ensuite à poser qqs définitions sur les notions de complexité.
Puis nous aborderons différentes méthodes utilisées en IA pour représenter et résoudre un pb. Ici, 3 formalismes de représentation seront présentés et traités :
- la logique,
- les graphes,
- les équations et inéquations mathématiques.

Ce cours est bâti en utilisant diverses références dont [Xuo92, AS93, GJ79, PS82, Van98].

2.2 Quelques exemples

On peut retrouver ici tous les problèmes de type "casse-tête" vus en théorie des graphes mais aussi des problèmes beaucoup plus pratiques.

Exemple 1 *[Coloration de carte/graphe] Etant donné une carte, il s'agit de colorer les régions de telle sorte que deux régions adjacentes (ayant une frontière commune) soient de couleurs différentes.*

Exemple 2 *[Les missionnaires et les cannibales] Trois missionnaires et trois cannibales se trouvent sur la rive droite d'une rivière. On dispose d'une barque sans passeur, pouvant emmener deux personnes au plus, et ne traversant jamais à vide. Il s'agit de faire traverser la rivière à toute la troupe. Attention : si les missionnaires se retrouvent sur une rive en nombre strictement inférieur au nombre de cannibales, ils se font manger.*

Exemple 3 *[Le jeu de Taquin 3×3] Ce jeu comporte 9 positions organisées en matrice 3×3. Huit positions sont occupées par des jetons*[1] *repérés chacun par un chiffre (de 1 à 8) ; la neuvième position, inoccupée, est appelée "case vide". Un jeton posé sur une case adjacente horizontalement ou verticalement à la case vide peut glisser sur la case vide.*[2] *C'est un mouvement élémentaire du jeu, et c'est le seul mouvement autorisé. On cherche une séquence de mouvements élémentaires permettant de passer d'une configuration initiale à une configuration but donnée (voir Figure 2.1).*

$$\begin{vmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & - & 5 \end{vmatrix} \qquad\qquad \begin{vmatrix} 1 & 2 & 3 \\ 8 & - & 4 \\ 7 & 6 & 5 \end{vmatrix}$$

FIGURE 2.1 – Etat initial, Etat-but d'un jeu de Taquin

Exemple 4 *[Die Hard] On dispose de deux récipients de 3 et 4 litres qu'on peut remplir, vider ou transvaser l'un dans l'autre. Le but est d'obtenir 2 litres dans l'un des deux.*

Exemple 5 *[Problème du sac à dos] On dispose d'un ensemble d'objets o_i avec $i = 1$ à 6, chacun ayant un poids et une valeur.*
- *poids des objets : $\{4, 3, 8, 4, 4, 9\}$ en kilos,*
- *valeurs des objets : $\{7, 2, 32, 9, 2, 8\}$ en euros.*

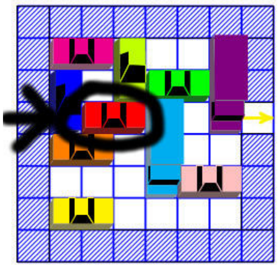
On veut les mettre dans un sac supportant au maximum un poids fixé (ici 23 kilos) en optimisant la valeur du contenu du sac.

Exemple 6 *[Planning du robot] Un robot doit exécuter un certain travail en plusieurs stations différentes. Dans chaque station, sont spécifiées les heures à partir desquelles on peut exécuter les tâches locales et les heures à ne pas dépasser. La durée d'exécution de chaque tâche est constante, les temps de transfert entre les différentes stations sont négligeables. On veut trouver une répartition des tâches pour le robot.*

Exemple 7 *[Déplacement multimodal] Il s'agit de trouver un itinéraire permettant de se rendre de l'UPS (IRIT) à la place Saint-Cyprien à Toulouse en moins de 45 minutes, en empruntant le métro, le bus et/ou le vélo.*

Exemple 8 *[Rush-Hour] Il s'agit de simuler une congestion automobile dans un parking à l'heure de pointe (*rush hour *en anglais). Le but du jeu est d'extraire le véhicule rouge d'une grille dans laquelle plusieurs autres véhicules bloquent la sortie. Il faut pour cela les déplacer (sans les soulever) en respectant les directions de déplacement imposées par la grille.*

1. On parle parfois de tuile au lieu de jeton.
2. On parlera de case pleine numérotée et d'échange d'une case pleine avec la case vide.



2.3 Classification des problèmes suivant leur type

Deux classifications sont possibles :
- soit en tenant compte du type de résultat attendu,
- soit en considérant l'existence d'un algorithme permettant d'atteindre ces résultats.

2.3.1 Classification suivant les résultats

Définition 1 (Problème de décision/d'optimisation) *Un problème est un problème :*
- *de* décision *(dit aussi de "reconnaissance" ou de "satisfaction") si la solution attendue est du type OUI ou NON,*
- *d'*optimisation *si la solution attendue est celle vérifiant un critère donné d'optimalité.*

Si on reprend chacun des exemples précédents, on voit très vite qu'ils peuvent correspondre à chacun de ces 2 types suivant la question (donc le problème) qu'on se pose.

Exemple 1 page précédente (cont'd)*[Coloration de carte/graphe] On peut se poser les questions suivantes :*
- *Existe-t-il une solution avec un nombre donné de couleurs ? (problème de décision)*
- *Trouver le nombre chromatique. (problème d'optimisation)*

Sur chacun des exemples 2 à 8, proposer des problèmes de décision et des problèmes d'optimisation.

Énoncé 1 *Example 2 sur les missionnaires et les cannibales. On peut se poser les questions suivantes :*
- *Existe-t-il une solution permettant de faire traverser tout le monde sans que personne ne se fasse manger ? (problème de décision)*
- *Existe-t-il une solution permettant de faire traverser tout le monde sans que personne ne se fasse manger en moins de 3 trajets de barque ? (problème de décision)*
- *Trouver une solution. (problème d'optimisation sans véritable critère à optimiser)*
- *Donner la solution la plus rapide en nombre de trajets de barque. (problème d'optimisation)*

Énoncé 2 *Example 3 sur le jeu de Taquin 3×3. On peut se poser les questions suivantes :*

- *Existe-t-il une solution pour passer de l'état de départ à l'état d'arrivée ? (problème de décision)*
- *Existe-t-il une solution pour passer de l'état de départ à l'état d'arrivée en moins de 10 mouvements élémentaires ? (problème de décision)*
- *Trouver une solution. (problème d'optimisation sans véritable critère à optimiser)*
- *Trouver la solution la plus courte en nombre de mouvements élémentaires. (problème d'optimisation)*

Énoncé 3 *Example 4 sur Die Hard. On peut se poser les questions suivantes :*
- *Existe-t-il une solution pour mettre x litres dans un des récipients pour $x = 2$? Pour $x = 1$? (problèmes de décision)*
- *Trouver une solution. (problème d'optimisation sans véritable critère à optimiser)*
- *Trouver la séquence de manipulations la plus courte à faire pour résoudre le problème qd $x = 2$ ou qd $x = 1$. (problèmes d'optimisation)*

Énoncé 4 *Example 5 sur le Problème du sac à dos. On peut se poser les questions suivantes :*
- *Existe-t-il un moyen de remplir complétement le sac (donc de mettre pour 23 kilos exactement dans le sac) ? (problème de décision)*
- *Existe-t-il un moyen de remplir complétement le sac (donc de mettre pour 23 kilos exactement dans le sac) avec moins de 5 objets ? (problème de décision)*
- *Trouver une solution. (problème d'optimisation sans véritable critère à optimiser)*
- *Comment remplir le sac en* maximisant *la valeur du contenu. (problème d'optimisation)*

Énoncé 5 *Example 6 sur le planning du robot. On peut se poser les questions suivantes :*
- *Existe-t-il une solution permettant d'enchaîner toutes les tâches demandées ? (problème de décision)*
- *Existe-t-il une solution permettant d'enchaîner toutes les tâches demandées en moins de 10 minutes ? (problème de décision)*
- *Trouver une solution. (problème d'optimisation sans véritable critère à optimiser)*
- *Trouver l'enchaînement des actions du robot pour qu'il exécute toutes les tâches demandées dans le temps le plus court possible. (problème d'optimisation)*

Énoncé 6 *Example 7 sur le déplacement multimodal. On peut se poser les questions suivantes :*
- *Existe-t-il une solution ? (problème de décision)*
- *Existe-t-il une solution sans utiliser le vélo ? (problème de décision)*
- *Trouver une solution. (problème d'optimisation sans véritable critère à optimiser)*
- *Trouver la meilleure solution en terme de temps ou en nb de changement de véhicule. (problème d'optimisation)*

Énoncé 7 *Example 8 sur Rush-Hour. On peut se poser les questions suivantes :*
- *Existe-t-il une solution ? (problème de décision)*
- *Existe-t-il une solution sans déplacer le camion bleu dont on a perdu la clé ? (problème de décision)*
- *Trouver une solution. (problème d'optimisation sans véritable critère à optimiser)*
- *Trouver la solution qui permet de déplacer le moins possible de voitures. (problème d'optimisation)*

Remarquons aussi qu'on peut souvent ramener un problème d'optimisation à un problème de décision. Par exemple :

Exemple 9 *Considérons l'exemple du voyageur de commerce (dit TSP pour "Travelling Salesman Problem") : soit un voyageur de commerce et un graphe non orienté représentant un ensemble de villes (les sommets) et les routes menant d'une ville à une autre (les arêtes), la tournée du voyageur de commerce doit passer dans toutes les villes une seule fois et être la plus courte possible. On a alors le problème de décision suivant : "Existe-t-il une tournée de longueur $\leq$ à une longueur k donnée ?".*
Sur cet exemple, on peut aussi avoir un problème d'optimisation avec plusieurs variantes possibles :
- *optimisation pure : trouver le circuit optimal,*
- *valeur optimale : trouver seulement la longueur du circuit optimal,*
- *témoin : trouver un circuit de longueur $\leq$ à une longueur k donnée.*

En traitant plusieurs valeurs de k, le problème de décision permet ainsi de traiter un des problèmes d'optimisation (celui de la valeur optimale).

2.3.2 Classification suivant l'existence d'un algorithme

Cette seconde catégorisation de problème n'existe que dans le cas d'un problème de décision.

Définition 2 (Problème décidable/indécidable) *Un problème de décision est dit décidable s'il existe un algorithme s'exécutant en un nombre fini d'étapes, qui le décide, c'est-à-dire qui réponde par OUI ou par NON à la question posée par le problème. S'il n'existe pas un tel algorithme, le problème est dit indécidable.*

Si on reprend le problème de coloration donné dans l'exemple 1 avec un graphe *fini*, le problème de décision associé est décidable : il suffit de construire puis de tester toutes les configurations d'affectation de couleur possibles pour un nb donné de couleurs ; le graphe étant fini, ce nombre de configurations l'est aussi et l'algorithme pourra donc donner sa réponse en un nb fini d'étapes (mais c'est évidemment un algorithme très inefficace).

Exemple 10 *Un exemple d'un problème indécidable est le* problème de l'arrêt *: il s'agit du problème de décision qui détermine, à partir de la description d'un programme informatique, si le programme s'arrête ou non. Voir aussi en Section 4.2.1.*
En effet, il n'existe pas de programme permettant de tester n'importe quel programme informatique[3] *afin de conclure* dans tous les cas *s'il s'arrêtera en un temps fini ou bouclera à jamais (preuve faite par Alan Turing en 1936).*

3. d'un langage suffisamment puissant, tels tous ceux qui sont utilisés en pratique.

Chapitre 3

Eléments de base sur la logique

Références bibliographiques : [AS93], [Ram88], [Gal86].
La logique étudiée ici est la logique des propositions (dite aussi logique propositionnelle, calcul des propositions, calcul propositionnel). C'est un langage qui se définit comme suit.

3.1 Syntaxe de la logique des propositions

On va donner ici quelques notions de base suffisantes pour pouvoir utiliser la logique dans le cadre de la programmation. Pour plus d'information, voir par exemple [AS93].

Vocabulaire Il s'agit des "mots" que l'on a le droit d'utiliser :

- un ensemble dénombrable de symboles $p_0, p_1, \ldots, p_n$ appelés *variables propositionnelles* (remarque : le n dans p_n devra être explicité),
- le symbole dénotant la *constante logique* FAUX : $\perp$,
- les symboles représentant les *connecteurs* binaires : $\wedge$ (appelé "et" ou "conjonction"), $\vee$ (appelé "ou" ou "disjonction"), $\rightarrow$ (appelé "implication"),
- les symboles *délimiteurs* : (et).

Syntaxe C'est-à-dire, quelles sont les "phrases" que l'on peut construire ? On appellera ces "phrases" des *formules (bien formées)* et on les définit à l'aide des règles suivantes :

1. Toute variable propositionnelle est une formule (dite *formule atomique*).
2. $\perp$ est une formule.
3. Si A et B sont des formules alors $(A \wedge B)$, $(A \vee B)$, et $(A \rightarrow B)$ sont des formules.
4. Les formules sont seulement les expressions obtenues par l'application des trois règles ci-dessus un nombre fini de fois.

Exemple : $(((p_0 \wedge p_1) \vee p_3) \rightarrow (p_1 \wedge p_2))$ est une formule.

Remarque importante : Il s'agit d'une définition volontairement abrégée puisqu'on n'y trouve pas la constante logique VRAI ($\top$), ni les connecteurs $\leftrightarrow$ et $\neg$. Malgré cela, la définition est complète puisque ces informations "manquantes" peuvent être calculées à partir des informations données dans la définition :

- $\neg A = A \rightarrow \perp$ (le connecteur unaire $\neg$ est appelé la négation),
- $\top = A \vee \neg A = \neg\perp$ (la constante logique $\top$ est appelée VRAI),
- $A \leftrightarrow B = (A \rightarrow B) \wedge (B \rightarrow A)$ (le connecteur binaire $\leftrightarrow$ est appelé l'équivalence).

3.2 Sémantique de la logique des propositions

3.2.1 Notations

On introduit une relation de priorité entre les connecteurs :

$$\neg > \wedge > \vee > \rightarrow > \leftrightarrow$$

Cette relation de priorité permet de supprimer les parenthèses dans les cas où il n'y a pas d'ambiguïté sur le connecteur à utiliser.
Toujours pour simplifier, on pourra omettre les parenthèses les plus externes.

Exemple : la formule $\neg A \wedge B \vee C \rightarrow D \leftrightarrow E$ signifie $(((((\neg A) \wedge B) \vee C) \rightarrow D) \leftrightarrow E)$.
Dans la pratique, on utilise surtout la priorité du $\neg$ et on supprime les parenthèses externes :

$$(((\neg A \wedge B) \vee C) \rightarrow D) \leftrightarrow E$$

3.2.2 Théorie des modèles

Etant donné qu'on se trouve en logique propositionnelle, une formule sera soit `vraie`, soit `fausse`.
D'autre part, il est évident que le sens (la valeur) d'une formule dépend des composantes de cette formule (donc de la valeur des sous-formules atomiques).
Soit V_P l'ensemble des symboles propositionnels du langage, et F l'ensemble des formules issues de V_P, on va donc définir ce qu'est une valuation.

Définition 3 (valuation) *L'application* $v \ : F \rightarrow \{$`vraie`, `fausse`$\}$ *sera une valuation si et seulement si :*

- $v(\top) =$ `vraie` *et* $v(\perp) =$ `fausse`,
- *soit A une formule du langage, $v(\neg A)$ respecte la table suivante :*

A	$\neg A$
vraie	fausse
fausse	vraie

- *soit A et B deux formules du langage et $\uparrow$ un connecteur binaire de la logique propositionnelle, $v(A \uparrow B)$ respecte les tables suivantes :*

A	B	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
vraie	vraie	vraie	vraie	vraie	vraie
vraie	fausse	fausse	vraie	fausse	fausse
fausse	vraie	fausse	vraie	vraie	fausse
fausse	fausse	fausse	fausse	vraie	vraie

Les tables données ici sont appelées tables de vérité.

Remarquons qu'une valuation est donc définie à partie de l'instanciation à `vrai` ou à `faux` de chaque symbole propositionel composant la formule valuée en respectant ensuite les tables de vérité.
Intuitivement :

- $A \wedge B$ est `vraie` si A et B sont toutes les deux `vraies` ;
- $A \vee B$ est `vraie` si A ou B sont `vraies` (une seule des deux suffit !) ;
- $A \rightarrow B$ est `vraie` soit parce que A est `vraie` et que B l'est aussi, soit parce que A est `fausse` ; on considère que $A \rightarrow B$ a la signification suivante en langage naturel *si A alors B* ;

voyons sur un exemple : Pluie $\rightarrow$ Parapluie (avec Pluie signifiant "il pleut" et Parapluie signifiant "je prends mon parapluie") ; dans ce cas, s'il pleut et que je prends mon parapluie alors la formule Pluie $\rightarrow$ Parapluie est `vraie` ; par contre, s'il pleut et que je ne prends pas mon parapluie alors la formule Pluie $\rightarrow$ Parapluie est `fausse` ; et enfin, s'il ne pleut pas, que je prenne ou pas mon parapluie n'a aucune importance et on décide donc par convention que la formule Pluie $\rightarrow$ Parapluie est `vraie` ;
- $A \leftrightarrow B$ est `vraie` si A et B ont la même valeur.

On peut constater sur ces tables de vérité que les connecteurs $\wedge$, $\vee$ et $\leftrightarrow$ sont commutatifs ($A \uparrow B = B \uparrow A$).

Définition 4 *Une formule A sera une* tautologie *ssi* $v(A) =$ `vraie`, $\forall v$. *On dira aussi que A est* valide *ou que A est un* théorème. *On notera alors* $\models A$.

Définition 5 *Une formule sera une* contradiction *ssi* $v(A) =$ `fausse`, $\forall v$. *On dira aussi que A est* invalide *ou* insatisfiable.

Définition 6 *Une formule A sera dite* consistante *ou* satisfiable *ssi* $\exists v$ *telle que* $v(A) =$ `vraie`.

Définition 7 *Une valuation v est un* modèle *de A ssi* $v(A) =$ `vraie`. *On dit que v* satisfait *A.*

Définition 8 *Une valuation v est un* contre-modèle *de A ssi* $v(A) =$ `fausse`. *On dit que v* falsifie *A.*

Définition 9 *Soit $A_1 \ldots A_n$ et B des formules du langage, on dira que B est une* conséquence logique *de $A1 \ldots A_n$ (noté $A_1, \ldots, A_n \models B$) ssi tout modèle de $A_1 \wedge \ldots \wedge A_n$ est aussi un modèle de B.*

Théorème 1 *[Thérorème de la déduction] Soit $A_1 \ldots A_n$ et B des formules du langage, on a :*

$$A_1, \ldots, A_n \models B \text{ ssi } \models (A_1 \wedge \ldots \wedge A_n) \rightarrow B$$

Exemples La formule $A \vee \neg A$ est une tautologie, la formule $A \wedge \neg A$ est une contradiction, alors que la formule $A \rightarrow \neg A$ est seulement consistante (la valuation $\{(A \rightarrow$ `vraie`$)\}$ étant un contre-modèle de $A \rightarrow \neg A$, alors que la valuation $\{(A \rightarrow$ `fausse`$)\}$ est un modèle de $A \rightarrow \neg A$) :

A	$\neg A$	$A \vee \neg A$	$A \wedge \neg A$	$A \rightarrow \neg A$
vraie	fausse	vraie	fausse	fausse
fausse	vraie	vraie	fausse	vraie

D'autre part, on constate sur cette table que $\neg A \models (A \rightarrow \neg A)$, de même que $(A \wedge \neg A) \models (A \rightarrow \neg A)$

Remarque La majorité des problèmes qu'on cherchera à résoudre en utilisant la logique désormais seront des *démonstrations de théorèmes*, c'est-à-dire montrer si une formule donnée est valide ou pas. Pour cela, on peut utiliser diverses techniques. Dans ce cours, on se contentera d'utiliser les tables de vérité : en énumérant la totalité des valuations de la formule (2^n valuations possibles pour une formule contenant n symboles propositionnels distincts) et en calculant pour chaque valuation la valeur de la formule.

3.3 Problèmes remarquables liés à la logique

Le principal problème utilisé dans ce cours est le problème SAT (pour "satisfiabilité d'une formule logique propositionnelle").

Définition 10 *Soit la formule logique propositionnelle Φ construite sur l'ensemble de symboles propositionnels $\{p_0, \ldots, p_n\}$. Le problème SAT consiste à répondre à la question*
"Φ est-elle satisfiable ?"
(c'est-à-dire, "existe-t-il un modèle de Φ ?", ou bien de manière équivalente, "existe-t-il une instanciation de toutes les variables p_i avec les valeurs soit `vrai`, *soit* `faux` *telle que Φ soit* `vraie` *?").*

Ce problème est clairement un problème de décision.
Il est intéressant de noter que des outils ont été spécialement développés pour traiter ce problème. Ces "solvers SAT" sont particulièrement efficaces quand la formule logique à vérifier est sous forme CNF ("Conjunctive Normal Form"), c'est-à-dire sous la forme d'une conjonction de disjonctions, les disjonctions ne contenant alors que des variables propositionnelles ou des négations de ces variables. Les disjonctions respectant cette caractéristique sont appelées des "clauses".
Considérons par exemple, les formules Φ_1, Φ_2 et Φ_3 suivantes (construites sur l'ensemble de symboles propositionnels $\{A, B, C\}$). Φ_1 est sous forme CNF alors que Φ_2 et Φ_3 ne le sont pas.

$$\Phi_1 = (\neg A \vee B \vee C) \wedge (B \vee C) \wedge A$$

$$\Phi_2 = (\neg A \wedge B) \wedge (B \vee C) \vee A$$

$$\Phi_3 = (\neg A \vee (B \wedge C)) \wedge (B \vee C) \wedge A$$

Notons qu'il est toutefois toujours possible de transformer une formule de la logique propistionnelle en une formule équivalente sous forme CNF.

3.4 Exercices

Énoncé 8 *Traduisez en logique propositionnelle les phrases suivantes :*

- *il pleut et il vente ;*
- *s'il pleut, je prends mon parapluie ;*
- *s'il ne pleut pas et qu'il n'y a pas de vent, je prends mon chapeau.*

Énoncé 9 *Traduisez en logique propositionnelle les phrases suivantes :*

- *on a un canari ;*
- *les canaris sont des oiseaux ;*
- *les oiseaux volent ;*
- *on a un manchot ;*
- *les manchots sont des oiseaux ;*
- *les manchots ne volent pas.*

Énoncé 10 *Vérifier si les formules suivantes (construites sur l'ensemble de symboles propositionnels $\{A, B, C\}$) sont des tautologies :*

1. $((A \wedge B) \wedge C) \leftrightarrow (A \wedge (B \wedge C))$,
2. $(A \wedge B) \leftrightarrow (B \wedge A)$,
3. $(A \wedge A) \leftrightarrow A$,
4. $(A \wedge (A \vee B)) \leftrightarrow A$,
5. $(A \wedge (B \vee C)) \leftrightarrow ((A \wedge B) \vee (A \wedge C))$,
6. $(A \wedge \neg A) \leftrightarrow \bot$,
7. $((A \vee B) \vee C) \leftrightarrow (A \vee (B \vee C))$,
8. $(A \vee B) \leftrightarrow (B \vee A)$,
9. $(A \vee A) \leftrightarrow A$,
10. $(A \wedge (A \vee B)) \leftrightarrow (A \wedge B)$,
11. $(A \vee (B \wedge C)) \leftrightarrow ((A \vee B) \wedge (A \vee C))$,
12. $(A \vee \neg A) \leftrightarrow \top$.

Énoncé 11 *Soit Φ la formule logique résultant de la conjonction des formules logiques issues de l'énoncé 8.*
Montrer que "je prends mon parapluie ou mon chapeau" est une conséquence logique de Φ.
Montrer que "si je prends mon parapluie alors je prends mon chapeau" n'est pas une conséquence logique de Φ.

Énoncé 12 *Soit Φ la formule logique résultant de la conjonction des 3 premières formules logiques issues de l'énoncé 9. Appliquez le problème SAT à Φ.*

Énoncé 13 *Soit Φ la formule logique résultant de la conjonction de toutes les formules logiques issues de l'énoncé 9. Appliquez le problème SAT à Φ.*

Bilan :
• Logique propositionnelle : langage formel (vocabulaire + syntaxe + sémantique)
• Notion de valuation et tables de vérité : une formule est soit `vraie`, soit `fausse`
• Problème SAT : est-ce qu'une formule donnée est satisfiable ? (*i.e.* est-ce qu'il y a un moyen (une valuation) de la rendre `vraie` ?)

Chapitre 4

Complexité

Il s'agit ici de donner les définitions de base concernant la notion de complexité.
Soit P un problème, et M une méthode pour résoudre le problème P, un algorithme A est la description de la méthode M dans un langage algorithmique.
La notion de complexité recouvre alors deux champs différents :
— soit la complexité des algorithmes (on cherche alors à mesurer l'efficacité de la méthode M pour résoudre P),
— soit la complexité des pbs (on cherche alors à mesurer la difficulté du pb P indépendamment d'une méthode de résolution particulière, donc la difficulté intrinsèque du problème).

4.1 Complexité algorithmique (dite pratique)

4.1.1 Définitions

Il s'agit de l'évaluation des ressources consommées par les algorithmes, en temps d'exécution et en espace mémoire. Ici, on ne se préoccupera que de l'aspect temporel[1].
C'est un moyen de comparaison des algorithmes (donc des méthodes qui sont "derrière" ces algos).
Sachant que le temps de calcul d'un programme dépend

1. des performances du processeur,
2. du compilateur utilisé,
3. des données en entrée du problème,
4. de l'algorithme lui-même,

il va falloir définir des métriques particulières qui vont pouvoir s'abstraire des 3 premiers éléments pour se consacrer uniquement au dernier. Puisqu'on veut évaluer independamment de la machine et tenir compte de données différentes, la taille des données en entrée sera donc un paramètre du temps de calcul.
Soit n la taille des données en entrée, on notera $T(n)$ le temps de calcul d'un algorithme en fonction de n. $T(n)$ correspondra au nombre d'opérations élémentaires réalisées en fonction de n.
Les notions à étudier concernant $T(n)$ seront alors :
— l'ordre de grandeur de $T(n)$ (le coût exact n'est pas intéressant, il suffit d'avoir un ordre de grandeur),
— son évolution,

1. La complexité spatiale correspond au nombre d'unités mémoires occupées lors de l'exécution de l'algorithme. Par exemple, pour un algorithme utilisant une matrice n sur m d'entiers, la complexité en espace sera au moins de $n \times m$ unités avec une unité correspondant à la place occupée par un entier.

— le pire cas (on peut aussi étudier le meilleur des cas[2] et le cas moyen[3]).
On utilisera la notation de Landau $O(f(n))$ qui caractérise le comportement asymptotique (*i.e.* quand n tend vers l'infini) :

$$T(n) = O(f(n)) \text{ si } \exists c, n_0 \text{ tels que } \forall n > n_0, T(n) \leq c \times f(n)$$

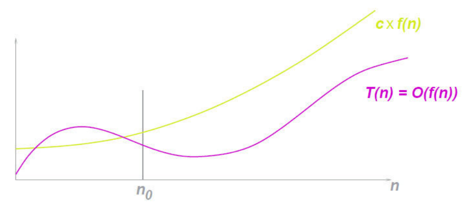
Trois autres notations sont possibles mais ne seront pas utilisées ici ; il s'agit des notations o (petit o, à ne pas confondre avec grand O correspondant à la notation de Landau), Θ et Ω :

$$T(n) = o(f(n)) \text{ si } \forall \epsilon > 0, \exists n_0 \text{ tel que } \forall n > n_0, T(n) \leq \epsilon \times f(n)$$

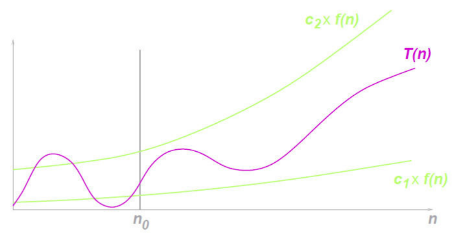
$$T(n) = \Theta(f(n)) \text{ si } \exists c_1, c_2, n_0 \text{ tels que } \forall n > n_0, c_1 \times f(n) \leq T(n) \leq c_2 \times f(n)$$

$$T(n) = \Omega(f(n)) \text{ si } \exists c, n_0 \text{ tels que } \forall n > n_0, 0 \leq c \times f(n) \leq T(n)$$

La notation O majore :

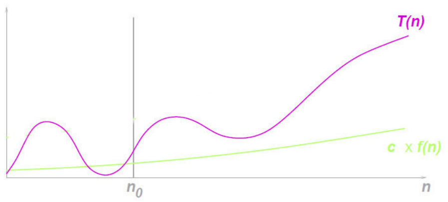


La notation o exprime le fait que T est négligeable devant f. Cette notation est utilisée en mathématiques mais très peu en informatique.
La notation Θ borne :



2. En général pas très intéressant car pas vraiment utilisable.
3. En général très difficile à faire car il faudrait disposer une distribution de probabilité décrivant les entrées.

La notation Ω minore :



Remarque : soit n un entier, on a l'échelle suivante :

$$log(n) < n < n \times log(n) < n^2 < n^3 < \ldots k^n < n! < n^n$$

La méthode utilisée pour calculer $T(n)$ consiste à étudier les différents éléments de l'algorithme :

Pour une séquence, on additionne : soit deux fragments I_1 et I_2, dont les complexités respectives sont $T_1(n)$ et $T_2(n)$, alors la complexité de la séquence $I_1; I_2$ est

$$T(n) = T_1(n) + T_2(n)$$

Pour un embranchement, on prend le max : soit la structure algorithmique *si (condition) alors* I_1 *sinon* I_2 avec $T_i(n)$ la complexité du fragment I_i et $T_C(n)$ celle du test, alors la complexité de la structure *si alors sinon* est

$$T(n) = T_C(n) + max(T_1(n), T_2(n))$$

Pour une boucle, on additionne le coût de chaque itération : soit la structure algorithmique *tantque (condition) I* avec k passages dans la boucle, $T_I(n)$ la complexité du fragment I et $T_C(n)$ celle du test, alors la complexité de la structure *tantque* est

$$T(n) = \Sigma_{i=1}^{k}(T_I(n) + T_C(n)) + T_C(n)$$

donc

$$T(n) \approx \Sigma_{i=1}^{k}(T_I(n) + T_C(n))$$

Remarque : on peut aussi évaluer des fragments définis par des équations récursives mais on ne le fera pas ici.
On dit qu'un algorithme est :
— en temps constant si $T(n) = O(1)$,
— logarithmique si $T(n) = O(log(n))$,
— linéaire si $T(n) = O(n)$,
— polynomial si $T(n) = O(n^k)$ (en particulier quadratique si $k = 2$),
— exponentiel si $T(n) = O(x^n)$ avec $x \geq 2$.

4.1.2 Exercices

Énoncé 14 *Soit le programme suivant, donnez sa complexité avec la notation O :*

```
// permutation de deux variables
tmp = x ;
x = y ;
y = tmp ;
```

Énoncé 15 *Soit le programme suivant, donnez sa complexité avec la notation O :*

```
// calcul d'un maximum
if (x < y) max = y ;
else max = x ;
```

Énoncé 16 *Soit le programme suivant, donnez sa complexité avec la notation O :*

```
// recherche séquentielle dans une collection (cases numérotées de 0 à n-1)
i = 0 ;
tant que ((i < n) et (S[i] != x)) faire
    i = i + 1 ;
```

Énoncé 17 *Soit le programme suivant, donnez sa complexité avec la notation O :*

```
// Calcul du nb d'éléments d'une collection vérifiant une condition
cptPair = 0 ;
for (i = 0 to n-1) do
   if (S[i]  est pair) alors
       cptPair ++ ;
```

Énoncé 18 *Soit le programme suivant, donnez sa complexité avec la notation O :*

```
// tri à bulle d'une collection (cases numérotées de 1 à n)
pour i = n à 2 faire // au pire (n - 1) fois
   pour j = 1 à i - 1 faire  // au pire (i - 1) fois
      si (S[j] > S[j + 1]) alors
         permuter S[j] et S[j + 1]
```

Énoncé 19 *Soit le programme suivant, donnez sa complexité avec la notation O :*

```
// tri par sélection d'une collection (cases numérotées de 1 à n)
for (i = 1 to n-1) do
   min = S[i] ;
   p = i ;
   for (j = i+1 to n) do
      if (S[j] < min) then
         min =S[j] ;
         p = j ;
      endif
      S[p] = S[i] ;
      S[i] = min ;
```


Énoncé 20 *Soit les deux programmes suivants qui mettent à jour chaque case i d'un tableau S2 avec la somme des valeurs de 0 à i du tableau S1. Donnez leur complexité avec la notation O et utilisez-la pour dire lequel est le plus efficace :*

```
// programme 1
for (i=0  à n-1) do
   S2[i] = 0 ;
   for (j = 0 à i) do
       S2[i] = S2[i] + S1[j] ;

// programme 2
S2[0] = S1[0] ;
for (i=1  à n-1) do
   S2[i] = S2[i-1] + S1[i] ;
```

4.1.3 Quelques chiffres

La complexité de quelques algos bien connus :
— recherche dichotomique en $O(log_2(n))$,
— tri par fusion [4] en $O(n \times log(n))$,
— recherche dans un arbre de recherche en $O(n)$ avec n le nb de niveaux de l'arbre (et pas le nb de valeurs stockées dans l'arbre),
— parcours en largeur d'un graphe en $O(nbSommets + nbArcs)$.

Le tableau ci-après donne une estimation du temps d'exécution en fonction de la complexité et de la taille des données (hypothèse : une opération élémentaire s'exécute en 1 μs). On voit bien que plus la complexité temporelle d'un algo est importante et plus le temps d'exécution devient prohibitif.

$T(n)$	n			
	10	20	40	60
$log(n)$	1 μs	1.3 μs	1.6 μs	1.8 μs
n	10 μs	20 μs	40 μs	60 μs
$n \times log(n)$	10 μs	26 μs	64 μs	107 μs
n^2	100 μs	400 μs	1.6 ms	3.5 ms
n^3	1 ms	8 ms	64 ms	216 ms
2^n	1 ms	1 s	13 jours	366 siècles
3^n	59 ms	58 mn	3855 siècles	1.3×10^{13} siècles

NB : l'âge de l'univers est estimé à 10^8 siècles.
Le tableau suivant montre le gain que l'on peut espérer avec des machines allant plus vite : plus la complexité temporelle d'un algo est importante et moins on peut espérer de gain.
Soit $N =$ la taille du problème qu'on peut résoudre aujourd'hui.

$T(n)$	aujourd'hui	si 100 fois plus vite	si 1000 fois plus vite
n	N	100N	1000N
n^2	N	10N	32N
n^3	N	4.6N	10N
2^n	N	N+7	N+10
3^n	N	N+4	N+6

4. C'est une variante de tri assez similaire au tri rapide vu en SRI L3 qui utilise la technique du "diviser pour régner".

4.2 Complexité des problèmes (dite théorique)

Il s'agit d'estimer la difficulté intrinsèque d'un problème (et pas d'une méthode particulière pour résoudre ce problème). Si on reprend par exemple le problème de l'énoncé 20 page précédente, on voit bien qu'un même problème peut être résolu par des méthodes plus ou moins consommatrices de temps (voire d'espace) et pourtant la difficulté du problème reste inchangée quelle que soit la méthode choisie.
De même, si on prend le problème du tri d'une liste d'entiers, on voit bien qu'il existe plein de méthodes différentes avec des complexités différentes.
L'étude de la complexité des problèmes permet de déterminer des classes de problèmes et de pouvoir ainsi donner une limite minimale à la complexité des algorithmes permettant de résoudre efficacement ces problèmes.
Cette classification s'établit en utilisant un outil particulier : la machine de Turing.

4.2.1 Machine de Turing

(*cf.* Alan Turing, mathématicien anglais ; ses travaux ont permis de casser le code d'Enigma, machine de cryptage utilisée par l'armée allemande lors de la seconde guerre mondiale)

En informatique théorique, une machine de Turing est un modèle abstrait du fonctionnement des appareils mécaniques de calcul, tel un ordinateur et sa mémoire. Ce modèle a été imaginé par Alan Turing en 1936 afin de donner une définition précise du concept d'algorithme ou de "procédure mécanique".
Une machine de Turing n'est pas vraiment une "machine", c'est un concept abstrait, c'est-à-dire un objet mathématique. Une machine de Turing est un automate comportant les éléments suivants :
— un ruban infini divisé en cases consécutives. Chaque case contient un symbole choisi dans un alphabet fini. L'alphabet contient un symbole spécial appelé "symbole blanc" (0 dans l'exemple qui suit), et un ou plusieurs autres symboles. Le ruban est supposé être de longueur infinie vers la gauche ou vers la droite, en d'autres termes la machine aura toujours assez de longueur de ruban pour son exécution [5]. On considère que les cases non encore écrites du ruban contiennent le symbole blanc ;
— une tête de lecture/écriture qui peut lire et écrire les symboles sur le ruban, et se déplacer vers la gauche ou vers la droite du ruban ;
— un registre d'état qui mémorise l'état courant de la machine de Turing. Le nombre d'états possibles est toujours fini, et il existe un état spécial appelé "état de départ" qui est l'état initial de la machine avant son exécution ;
— une table d'actions qui indique à la machine quel symbole écrire sur le ruban, comment déplacer la tête de lecture (par exemple "G" pour une case vers la gauche, "D" pour une case vers la droite), et quel est le nouvel état, en fonction du symbole lu sur le ruban et de l'état courant de la machine. Si aucune action n'existe pour une combinaison donnée d'un symbole lu et d'un état courant, la machine s'arrête sur un échec.

À chaque étape de son calcul, la machine évolue en fonction de l'état dans lequel elle se trouve, et du symbole inscrit dans la case du ruban où se trouve la tête de lecture. Ces deux informations permettent la mise à jour de l'état de la machine grâce à la fonction de transition décrite dans la table des actions. À l'instant initial, la machine se trouve dans l'état initial, et le mot inscrit sur le ruban est l'entrée du programme. La machine s'arrête avec succès lorsqu'elle rentre dans un état terminal. Le résultat du calcul est alors le mot inscrit sur le ruban.

5. Ce qui montre bien que ce n'est pas une machine "réelle".

La notion d'algorithme correspond ainsi à la définition exacte du comportement de l'automate en fonction du ruban. Sa complexité en temps est le nombre d'opérations qu'il doit effectuer avant de terminer avec succès.

Exemple 11 *La machine de Turing qui suit possède un alphabet* $\{0, 1\}$, 0 *étant le "blanc". On suppose que le ruban contient une série de* 1, *et que la tête de lecture/écriture se trouve initialement au-dessus du* 1 *le plus à gauche. Cette machine a pour effet de doubler le nombre de* 1, *en intercalant un* 0 *entre les deux séries. Par exemple,* 111 *devient* 1110111. *L'ensemble d'états possibles de la machine est* $\{e_1, e_2, e_3, e_4, e_5\}$ *et l'état initial est* e_1.
La table d'actions est la suivante :

Ancien état	*Symbole lu*	*Symbole écrit*	*Mouvement*	*Nouvel état*
e_1	0	*(Arrêt)*		
	1	0	*Droite*	e_2
e_2	1	1	*Droite*	e_2
	0	0	*Droite*	e_3
e_3	1	1	*Droite*	e_3
	0	1	*Gauche*	e_4
e_4	1	1	*Gauche*	e_4
	0	0	*Gauche*	e_5
e_5	1	1	*Gauche*	e_5
	0	1	*Droite*	e_1

L'exécution de cette machine pour une série de deux 1 *serait (la position de la tête de lecture/écriture sur le ruban est inscrite en caractères gras, rouges et soulignés) :*

Exécution (1)			*Exécution (2)*			*Exécution (3)*			*Exécution (4)*		
Étape	*État*	*Ruban*	*Étape*	*État*	*Ruban*	*Étape*	*État*	*Ruban*	*Étape*	*État*	*Ruban*
1	e_1	$\underline{1}1$	*5*	e_4	$01\underline{0}1$	*9*	e_2	$10\underline{0}1$	*13*	e_4	$10\underline{0}11$
2	e_2	$0\underline{1}$	*6*	e_5	$0\underline{1}01$	*10*	e_3	$100\underline{1}$	*14*	e_5	$1\underline{0}011$
3	e_2	$01\underline{0}$	*7*	e_5	$\underline{0}101$	*11*	e_3	$1001\underline{0}$	*15*	e_1	$11\underline{0}11$
4	e_3	$010\underline{0}$	*8*	e_1	$1\underline{1}01$	*12*	e_4	$100\underline{1}1$		*(Arrêt)*	

Le comportement de cette machine peut être décrit comme une boucle :

- *Elle démarre son exécution dans l'état* e_1, *remplace le premier* 1 *par un* 0.
- *Puis elle utilise l'état* e_2 *pour se déplacer vers la droite, en sautant les* 1 *(un seul dans cet exemple) jusqu'à rencontrer un* 0, *et passer dans l'état* e_3.
- *L'état* e_3 *est alors utilisé pour sauter la séquence suivante de* 1 *(initialement aucun) et remplacer le premier* 0 *rencontré par un* 1.
- *L'état* e_4 *permet de revenir vers la gauche jusqu'à trouver un* 0, *et passer dans l'état* e_5.
- *L'état* e_5 *permet ensuite à nouveau de se déplacer vers la gauche jusqu'à trouver un* 0, *écrit au départ par l'état* e_1.
- *La machine remplace alors ce* 0 *par un* 1, *se déplace d'une case vers la droite et passe à nouveau dans l'état* e_1 *pour une nouvelle itération de la boucle.*

Ce processus se répète jusqu'à ce que e_1 *tombe sur un* 0 *(c'est le* 0 *du milieu entre les deux séquences de* 1*) ; à ce moment, la machine s'arrête avec succès (puisqu'elle arrive dans l'état terminal* e_1 *avec* 0 *sur le ruban en entrée).*

Rappelons que le *problème de l'arrêt* évoqué en section 2.3.2 comme l'archétype des problèmes indécidables a été exprimé par Alan Turing en 1936 sous la forme suivante : il s'agit du problème

de décision qui détermine, à partir de la description d'une machine de turing quelconque et de son ruban d'entrée, si la machine s'arrête ou non.

On distingue deux sortes de machine de Turing (donc deux types d'algorithmes) :

déterministe : à chaque instant, la machine est dans un certain état et l'action qu'elle effectue ne dépend que de cet état. (cela correspond +/- à l'ordinateur moderne idéalisé)

non déterministe : il existe une instruction de choix non déterministe pour déterminer l'état suivant.

Donc, étant donné le caractère lu sur le ruban et l'état courant, une machine de Turing déterministe a au plus une transition possible, alors qu'une machine de Turing non déterministe peut en avoir plusieurs. En conséquence, les calculs d'une machine de Turing déterministe forment une suite, alors que ceux d'une machine de Turing non déterministe forment un arbre, dans lequel chaque chemin correspond à une suite de calculs possibles.
On peut se représenter l'évolution d'une machine de Turing non déterministe ainsi : dans un état où il y a plusieurs transitions possibles, elle se duplique (triplique, etc.) et une sous-machine de Turing est créée pour chaque transition différente.

4.2.2 Classification des problèmes suivant leur complexité

Si on veut classifier les problèmes d'un point de vue complexité, il nous faut utiliser un outil identique pour tous, et cet outil, ce sera la machine de Turing : la complexité d'un problème est définie comme étant celle de l'algorithme exécuté sur la machine de Turing idéale le résolvant. Toutefois les machines de Turing ne peuvent traiter que des problèmes de décision (elles s'arrêtent sur un échec ou un succès). Les classes de complexité ainsi définies n'auront de sens que pour des problèmes de décision.

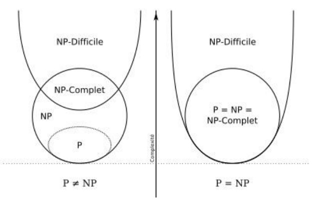
Définition 11 (La classe P) *C'est la classe des problèmes qui admettent un algorithme déterministe avec un temps d'exécution polynomial en fonction de la taille du ruban d'entrée.*

Définition 12 (La classe NP) *C'est la classe des problèmes qui admettent un algorithme non-déterministe avec un temps d'exécution polynomial en fonction de la taille du ruban d'entrée.*

Attention : NP veut donc dire "non-deterministe polynomial" et pas "non polynomial" puisqu'on a de façon évidente P $\subseteq$ NP (un algo déterministe étant un algo non déterministe avec au plus une seule transition possible pour chaque état).
La question à 1 million de $ est : a-t-on NP = P ?
Etant donné que tout pb de P est aussi dans NP, il peut être intéressant de caractériser les problèmes de NP qui ne peuvent pas être dans P. Cela correspond à la notion de problème NP-complet :

Définition 13 *Un problème est dit NP-complet s'il appartient à NP et s'il est aussi "difficile" que n'importe quel problème de NP.*

Le schéma suivant donne les relations d'inclusion entre classes de complexité :



Certains problèmes ont déjà été démontrés comme étant des problèmes NP-complets et donc, quand on a un nouveau problème à étudier du point de vue complexité, il suffit de montrer que ce nouveau problème se ramène aux problèmes déjà connus (il existe plusieurs méthodes mais on ne traitera pas ce type de problématique dans ce cours).

Exemple 12 *Un exemple emblématique de problème de la classe NP est le problème SAT (voir section 3). Il s'agit d'un problème de décision. Et il est aussi NP-complet.*
Prenons par exemple la formule logique propositionnelle $\Phi = \neg(x_1 \vee x_2)$. *Le problème SAT consiste à répondre à la question "*Φ *est-elle satisfiable ?". Il existe deux algorithmes possibles pour résoudre ce pb :*

- *Un algo déterministe : tester toutes les instanciations des* n *variables propositionnelles (*$O(2^n)$*) et vérifier si la formule est vraie (en* $O(m)$, *taille de la formule) pour l'une de ces instanciations. Cela revient à faire la table de vérité (pour* n *variables, il existe* 2^n *instanciations possibles) :*

x_1	x_2	Φ
V	*V*	*F*
V	*F*	*F*
F	*V*	*F*
F	*F*	*V*

  Et dans le pire des cas, ce sera la dernière instanciation qui nous donnera la solution. La complexité de cet algo est donc en $O(2^n)$.
- *Un algo non déterministe : pour toute variable* x_i, *le choix consiste soit à affecter la variable à vrai, soit à l'affecter à faux. Quand c'est fini, on teste la formule (si vrai alors succès). Donc la complexité de l'algo non déterministe est* $< (C_1 \times n) + (C_2 \times m)$ *donc en* $O(m+n)$. *En fait l'algo non-déterministe est similaire en complexité à celui, déterministe, consistant à vérifier qu'une instanciation du problème est solution. On dira donc que SAT est de complexité NP.*

Exemple 9 page 7 (cont'd) *Si on reprend l'exemple du voyageur de commerce, en terme de complexité, le problème de décision est NP-complet et on a aussi la relation de comparaison entre les complexités des différents problèmes :*

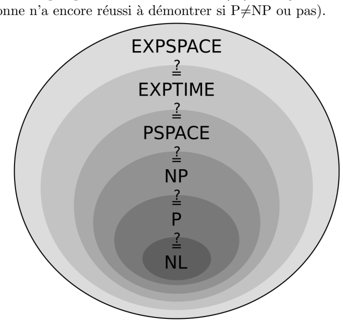
décision $\leq$ *témoin, valeur optimale* $\leq$ *optimisation pure.*

Remarquons qu'il existe toute une hiérarchie de classes de complexité pour les problèmes, certaines ne faisant intervenir que le temps et d'autres l'espace. En voilà, qq-unes :

- Classe L : c'est la classe des problèmes de décision qui peuvent être résolus par un algorithme déterministe en *espace* logarithmique par rapport à la taille de l'instance.
- Classe NL : cette classe correspond à la précédente mais pour un algorithme non-déterministe.

- Classe P : un problème de décision est dans P s'il peut être décidé par un algorithme déterministe en un *temps* polynomial par rapport à la taille de l'instance. On qualifie alors le problème de polynomial.
- Classe NP : c'est la classe des problèmes de décision pour lesquels la réponse oui peut être décidée par un algorithme non-déterministe en un *temps* polynomial par rapport à la taille de l'instance.
- Classe Co-NP : nom parfois donné pour l'équivalent de la classe NP avec la réponse non.
- Classe PSPACE : les problèmes décidables par un algorithme déterministe en *espace* polynomial par rapport à la taille de son instance.
- Classe EXPTIME : les problèmes décidables par un algorithme déterministe en *temps* exponentiel par rapport à la taille de son instance.

Il y a des liens entre complexité en espace et complexité en temps. Le schéma suivant montre les relations d'inclusion entre quelques-unes de ces classes (il y a toujours un point d'interrogation dû au fait que personne n'a encore réussi à démontrer si P$\neq$NP ou pas).



Bilan :
- Complexité algorithmique *vs* complexité des problèmes
- Complexité temporelle *vs* complexité spatiale
- Complexité algorithmique : estimation temps d'exécution (notation O)
- Complexité des problèmes : classes de complexité et types de problème
  - Classe P : problèmes (de décision) faciles à résoudre
  - Classe NP : problèmes (de décision) faciles à vérifier
  - NP = non déterministe polynomial
  - Montrer qu'un problème est NP ne signifie pas qu'il n'est pas dans P
  - Montrer qu'un problème est dans NP est souvent une étape avant de montrer qu'il est NP-complet/ NP-difficile.

Chapitre 5

La logique : outil de représentation et résolution

Le premier langage de représentation de problème qu'on va utiliser dans ce cours est la logique. Il en existe plusieurs mais nous allons nous contenter de la logique propositionnelle (voir section 3) et montrer sur des exemples comment on peut modéliser un problème et le résoudre.
L'idée va donc être de modéliser un problème sous la forme d'une formule logique. Etant donné que cette logique permet essentiellement de manipuler des valuations correspondant à `vrai` ou `faux`, les problèmes ainsi modélisés seront donc des problèmes de décision.

5.1 Premier exemple : l'oiseau Titi vole-t-il ?

Etape 1 : description du problème Nous sommes dans un monde où Titi est un oiseau et les oiseaux volent. Notre problème : Peut-on déduire que Titi vole ?

Etape 2 : encodage en logique propositionnelle On va utiliser le vocabulaire suivant : O (oiseau), V (vole). La formule Φ représentant notre monde est la suivante :

$$\Phi = O \wedge (O \rightarrow V)$$

Etape 3 : résolution du problème On veut montrer que V peut se déduire de Φ, donc que V est une conséquence logique de Φ (au sens de la définition 9 page 11) : $\Phi \models V$. D'après le théorème de la déduction 1 page 11, cela revient à montrer que la formule $\Phi \rightarrow V$ est valide. Et donc que sa négation est toujours fausse, donc non satisfiable[1].
On va donc utiliser le problème SAT pour résoudre notre problème de décision : $\Phi \wedge \neg V$ est-elle satisfiable ?
Réponse : NON, donc on peut déduire que Titi vole

5.2 Deuxième exemple : le manchot Titi vole-t-il ?

Etape 1 : description du problème Nous sommes dans un monde où Titi est un manchot, les manchots sont des oiseaux, les oiseaux volent mais les manchots ne volent pas. Notre problème : Peut-on déduire que Titi vole ?

1. Sachant que la négation de $\Phi \rightarrow V$ ($\neg(\Phi \rightarrow V)$) est aussi égale à $\neg(\neg\Phi \vee V)$ et donc à $\Phi \wedge \neg V$, nous remarquons que cela correspond exactement à un raisonnement par l'absurde.

Etape 2 : encodage en logique propositionnelle On va utiliser le vocabulaire suivant : O (oiseau), V (vole), M (manchot). La formule Φ représentant notre monde est la suivante :

$$\Phi = M \wedge (M \rightarrow O) \wedge (O \rightarrow V) \wedge (M \rightarrow \neg V)$$

Notons que Φ est sous forme CNF.

Etape 3 : résolution du problème Idem premier exemple : on applique SAT. On va donc se poser la question : $\Phi \wedge \neg V$ est-elle satisfiable ?
Réponse : NON, donc on peut déduire que Titi vole.
Par contre, on peut aussi se poser la question inverse : est-ce qu'à partir de Φ je peux déduire $\neg V$. Cela correspond à $\Phi \wedge V$ est-elle satisfiable ?
Et ici aussi la réponse est NON, donc on peut déduire que Titi ne vole pas.
Ceci est un exemple d'une formule inconsistante à partir de laquelle on peut déduire n'importe quoi. Et cela montre la limite de la représentation par la logique propositionnelle : il faut que le monde représenté soit parfaitement cohérent !

5.3 Troisième exemple : coloration de graphe

Reprenons l'exemple 1 page 4.

Etape 1 : description du problème Soit le graphe $G = (X, E)$ avec $X = \{a, b, c\}$ et $E = \{(a, b), (b, c), (c, a)\}$. Nous allons traiter ici 2 problèmes :
- Pb_1 : Peut-on colorer G avec 3 couleurs ?
- Pb_2 : Peut-on colorer G avec 2 couleurs ?

Etape 2 : encodage en logique propositionnelle Il faut commencer par choisir le vocabulaire. Ici, il va falloir pour chaque sommet dans le graphe x variables (1 pour chaque couleur).

Prenons par exemple le problème Pb_1 et les couleurs R (rouge), B (bleu) et J (jaune), on va alors avoir les variables propositionnelles R_i, B_i et J_i pour $i = a, b, c$ (la signification de R_a est "a est coloré en rouge" ; idem pour les autres symboles).
Ensuite, il faut donner les contraintes. Première contrainte, un sommet doit avoir une et une seule couleur :
$\Phi_i = (R_i \wedge \neg B_i \wedge \neg J_i) \vee (\neg R_i \wedge B_i \wedge \neg J_i) \vee (\neg R_i \wedge \neg B_i \wedge J_i)$ pour $i = a, b, c$
Cette contrainte est correcte mais elle n'est pas sous forme CNF. Il va donc être plus judicieux de l'exprimer plutôt sous la forme :
$\Phi_i = (R_i \vee B_i \vee J_i) \wedge (\neg R_i \vee \neg B_i) \wedge (\neg R_i \vee \neg J_i) \wedge (\neg B_i \vee \neg J_i)$ pour $i = a, b, c$
Seconde contrainte, deux sommets adjacents ne peuvent pas avoir la même couleur :
pour l'arête (a, b), on a donc $\Psi_{ab} = (\neg R_a \vee \neg R_b) \wedge (\neg B_a \vee \neg B_b) \wedge (\neg J_a \vee \neg J_b)$
pour l'arête (b, c), on a donc $\Psi_{bc} = (\neg R_c \vee \neg R_b) \wedge (\neg B_c \vee \neg B_b) \wedge (\neg J_c \vee \neg J_b)$
pour l'arête (c, a), on a donc $\Psi_{ca} = (\neg R_a \vee \neg R_c) \wedge (\neg B_a \vee \neg B_c) \wedge (\neg J_a \vee \neg J_c)$
Ici, toutes les formules sont bien sous forme CNF.

Dans le cas du problème Pb_2, le vocabulaire est réduit aux variables propositionnelles R_i, B_i et on aura les formules suivantes :
$\Phi_i = (R_i \vee B_i) \wedge (\neg R_i \vee \neg B_i)$ pour $i = a, b, c$
$\Psi_{ab} = (\neg R_a \vee \neg R_b) \wedge (\neg B_a \vee \neg B_b)$
$\Psi_{bc} = (\neg R_c \vee \neg R_b) \wedge (\neg B_c \vee \neg B_b)$
$\Psi_{ca} = (\neg R_a \vee \neg R_c) \wedge (\neg B_a \vee \neg B_c)$

Etape 3 : résolution du problème Ici aussi, pour Pb_1 ou Pb_2, on va utiliser SAT en posant la question : soit $\Phi = \Phi_a \wedge \Phi_b \wedge \Phi_c \wedge \Psi_{ab} \wedge \Psi_{bc} \wedge \Psi_{ca}$, est-ce que Φ est satisfiable ?
Si la réponse est oui, cela signifie qu'on peut trouver une assignation avec le nb de couleurs correspondant à la question posée pour le graphe G.

5.4 Quatrième exemple : sudoku

Etape 1 : description du problème Soit la grille suivante de sudoku (chaque case étant repérée par sa position dans la grille : numéro de ligne de 1 à 9 et numéro de colonne de 1 à 9) :

8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		

AI14473 (c) Arto Inkala www.aisudoku.com

☆☆☆☆☆☆☆☆☆☆☆

Notre problème : Peut-on placer la valeur 1 dans la case (2,2) ?

Etape 2 : encodage en logique propositionnelle Le principe d'encodage est le même que celui utilisé dans l'exemple précédent mais avec beaucoup plus de variables. En effet, il nous faut une variable pour chaque valeur possible dans chaque case : V_{ijx} pour i numéro de ligne, j numéro de colonne et x valeur possible (donc $i, j, x \in [1..9]$).
Puis, il faut donner toutes les contraintes :
- sur chaque ligne, pas deux fois la même valeur,
- sur chaque colonne, pas deux fois la même valeur,
- dans chaque bloc, pas deux fois la même valeur,
- dans chaque case, une seule valeur au maximum.

Par exemple, la première contrainte pourrait être exprimée à l'aide de la formule CNF suivante :

$$\forall i = 1 \ldots 9, \forall j = 1 \ldots 9, \forall k = 1 \ldots 9 ((j \neq k \wedge V_{ijx}) \rightarrow \neg V_{ikx})$$

Il faut aussi dire qu'une case ne peut prendre qu'une seule valeur :

$$\forall i = 1 \ldots 9, \forall j = 1 \ldots 9, \forall x = 1 \ldots 9, \forall y = 1 \ldots 9 ((x \neq y \wedge V_{ijx}) \rightarrow \neg V_{ijy})$$

Il faut ensuite décrire le contenu de la matrice, donc de chaque case déjà occupée. Par exemple :

$$V_{118} \wedge V_{233} \wedge V_{246} \wedge \ldots$$

Etape 3 : résolution du problème Considérons la conjonction Φ de toutes les formules précédentes, ainsi que la variable V_{221}. On utilise à nouveau SAT pour répondre à la question $\Phi \wedge V_{221}$ est-elle satisfiable ?
Si c'est le cas, cela signifie que mettre 1 dans la case $(2, 2)$ ne viole aucune des contraintes et est donc possible.
Remarquons que pour résoudre le sudoku complètement, il faut rajouter des contraintes supplémentaires disant qu'il faut que, pour chaque case (i, j), une des variables V_{ijx} soit `vraie`, pour $x = 1, \ldots, 9$.

Notons qu'on aurait pu choisir de ramener le problème de résolution du Sudoku à un problème de coloration de graphe en définissant le graphe suivant :
- chaque case de la grille est un sommet du graphe ;
- chaque case est liée aux 8 autres cases de la même ligne, aux 8 autres cases de la même colonne et aux 4 autres cases non encore atteintes dans le même bloc (c-à-d celles qui ne sont pas sur la même ligne ou la même colonne) ; cela fait donc 20 arêtes partant de chaque sommet ;
- chaque valeur de case possible correspond à une couleur ; on a donc 9 couleurs possibles.

Le problème courant aurait donc été une variante du pb de coloration dans lequel une coloration partielle des sommets est proposée et il faut savoir si elle satisfait les contraintes liées aux arêtes.

Bilan :
- Représentation d'un problème sous la forme d'une formule logique
- Résolution en se ramenant au problème SAT
- Attention, à la cohérence des formules ! Sinon, résultats sans intérêt.

Chapitre 6

Espaces d'états

Le second type de formalisation des problèmes est l'utilisation d'un *graphe*, appelé *espace d'états*, dans lequel chaque sommet est un *état du problème* et chaque arc est un moyen de passer d'un état à un autre état.

6.1 Formalisation d'un problème par espace d'états

6.1.1 Introduction

On s'intéresse à des problèmes pour lesquels il existe une description formelle, mais pas de méthode de résolution spécifique et dont l'environnement est supposé observable et déterministe. En général la meilleure façon de résoudre un tel problème n'est pas connue.
Reprenons par exemple quelques problèmes évoqués précédemment.

Exemple 6 page 4 (cont'd)*[Planning du robot] Considérons le problème d'ordonnancement (organisation dans le temps) des tâches d'un robot. Une bonne solution consiste à minimiser les ressources consommées par le robot (par exemple, le temps). Aller vers la tâche la plus urgente est un exemple de méthode simple, mais ce n'est pas nécessairement une bonne méthode. Dans le cas général, ce problème conduit à une explosion combinatoire dépendant du nombre de stations que devra faire le robot.*

Exemple 7 page 4 (cont'd)*[Déplacement multimodal] Le monde réel est trop complexe pour être modélisé. On ne détaillera pas les actions permettant de se rendre de l'IRIT à la station de métro (longer le bâtiment 1A, puis tourner à gauche, ...). On représentera ce trajet par une action abstraite "marcher jusqu'à la station UPS" dont le coût peut être estimé à 5 minutes. Une solution pourra donc être décrite par : "Marcher IRIT → station UPS ; Métro ligne B → station Carmes ; Marcher station Carmes → station Esquirol ; Métro ligne A → Saint-Cyprien". Une autre solution serait : "Marcher IRIT→ station UPS ; Métro ligne B →station Jean-Jaurès ; Métro ligne A → Saint-Cyprien."*

Le formalisme des espaces d'états permet de représenter un problème grâce aux notions d'état et d'opérateur.

6.1.2 Etats et opérateurs

Un problème peut être décrit par différentes variables.
- Un **état** du problème est l'ensemble des valeurs des variables, à un instant donné.
- **L'espace d'états** est l'ensemble de tous les états possibles du problème. On distingue un état dit **état initial**.
- Une **solution** (ou état-but) du problème est un état ayant des caractéristiques particulières.

- La description du problème fait aussi apparaître des **opérateurs**, permettant de transformer un état en un autre état.

Un problème est donc caractérisé par :
— la description d'un état possible,
— un état initial,
— une description d'un état-but (explicite ou implicite par ses propriétés),
— la description des opérateurs.

On appelle descendant d'un état s, tout état accessible en appliquant une séquence non vide d'opérateurs à s. Dans le cas d'une séquence réduite à un seul opérateur, il s'agit d'un successeur (ou descendant immédiat ou encore fils) de s. L'ensemble de tous les états accessibles à partir de l'état initial s'appelle **l'espace de recherche**. On représente généralement l'espace de recherche sous la forme d'un graphe d'états ou sous la forme d'une arborescence (sommet = état et arc d'un sommet x vers un sommet y = application d'un opérateur sur x conduisant à y ; voir exemples ci-dessous).

La **résolution** d'un problème est la recherche (et la découverte si le problème admet une solution) d'un état-but dans l'espace de recherche.

Remarque 1 *Les contraintes liées au problème sont prises en compte dans l'applicabilité des opérateurs : un opérateur n'est pas toujours applicable (ou autorisé) à tout état possible du problème.*
La résolution du problème a pour variante la recherche d'une séquence d'opérateurs (ou chemin) conduisant à un état-but. La distinction entre recherche d'un chemin et recherche d'un état-but n'est pas importante théoriquement, car le chemin peut être inclus dans l'état.
La résolution du problème peut aussi exiger l'obtention d'une solution optimale dans un sens particulier (selon un critère de comparaison des solutions). Ce problème d'optimisation est plus complexe.

Exemple 6 page 4 (cont'd)
Etat possible = planning codé par une liste de paires (station, heure d'arrivée)
Etat initial = planning vide
Etat-but = planning complet avec respect des échéances
Opérateur = ajout d'une paire au planning courant
Dans ce problème, on recherche un état-but.

6.2 Exercices

Proposer une représentation par espace d'état permettant de raisonner sur les problèmes suivants :
— Les missionnaires et les cannibales (voir exemple 2),
— Le jeu de Taquin (voir exemple 3),
— La coloration (voir exemple 1),
— Le problème de Die-Hard (voir exemple 4, dérouler uniquement 2 niveaux dans le graphe),
— Le problème du sac à dos (voir exemple 5)

Bilan :
Représentation d'un problème par un espace d'états avec :
• un état initial,
• des états buts (différents suivant le type de problème, décision ou optimisation),
• des opérateurs pour passer d'un état à un autre,
• certains états peuvent être interdits (contraintes sur les états ou les opérateurs),
• grâce aux opérateurs, on ne construit pas tout l'espace,
• trouver une solution au problème consiste à parcourir l'espace de recherche (de l'état initial vers un état but).

Chapitre 7

Méthodes complètes

L'exploration d'un espace d'état pour trouver une solution répondant à un certain critère est un problème d'optimisation et le problème de décision correspondant est en général NP-complet. On a donc affaire à des problèmes difficiles à résoudre au sens de la complexité.
Nous présentons tout d'abord les principes de la résolution d'un problème par la recherche d'une solution dans l'espace de recherche du problème. Nous rappelons ensuite les principales stratégies existantes (non informées puis informées).

7.1 Les principes

La résolution de problèmes pourrait donc se résumer à l'exploration de l'espace de recherche. Cependant, dans la plupart des cas, on se heurte à un problème d'explosion combinatoire : par exemple, dans le cas du Taquin de taille n, l'espace de recherche comporte $\frac{(n \times n)!}{2}$ états ; ce qui donne, pour $n = 3$, 181440 états).
Des techniques classiques consistent à construire **au fur et à mesure** une partie de l'espace de recherche en utilisant des stratégies d'exploration pour restreindre les états à considérer. Une stratégie définit le choix de l'état à considérer pour poursuivre la recherche.

Nous utiliserons dans la suite le vocabulaire suivant :
— Un état est **exploré** (ou **développé**) si on crée (ou génère) ses successeurs.
— Un état est **créé** (ou **généré**) lorsqu'il est produit par application d'un opérateur à un autre état. On calcule alors le code de sa représentation. Par convention, l'état initial est créé.
— Lors d'une exploration de l'espace de recherche, les états développés seront rangés dans l'ensemble appelé **Vus**, les états créés mais non encore développés seront rangés dans l'ensemble appelé **EnAttente**.

La mise en oeuvre d'une stratégie d'exploration produit un algorithme de recherche qui peut être évalué par différents critères :

Complétude Si le problème admet une solution, l'algorithme s'arrête en fournissant une solution.

Complexité temporelle Temps nécessaire pour trouver une solution.

Complexité spatiale Place mémoire nécessaire pour effectuer l'exploration.

Admissibilité Ce critère (encore appelé optimalité) s'applique à la recherche d'une solution avec coût. Une recherche est admissible si elle fournit une meilleure solution.

Rappelons que la complexité (temporelle ou spatiale) est toujours relative à une mesure de difficulté du problème. En informatique théorique, la mesure typique est la taille du graphe de l'espace des états (nombre de sommets + nombre d'arêtes, voir chapitre 4 page 15).
Ici, on ne va pas créer le graphe ; il sera représenté implicitement par son état initial et les opérateurs. On exprime donc la complexité en fonction de trois valeurs :
— facteur de branchement (b = nombre maximum de fils d'un état),
— profondeur maximum de l'arbre de recherche (m),
— profondeur d'un état-but le moins éloigné (d).
On peut considérer que la profondeur d'un nœud dans un arbre est la longueur du chemin menant du nœud racine à ce nœud (longueur comptée en nombre de nœuds). On rappelle alors que la taille maximale de l'arbre de recherche, notée n, est bornée de la manière suivante :

$$m \leq n \leq \frac{b^m - 1}{b - 1}$$

Donc pour un arbre de profondeur maximum 4 et de facteur de branchement 3, il y aura au maximum 40 états dans l'arbre de recherche, répartis en 4 niveaux contenant respectivement 1 ($= 3^0$ sur le niveau 1), 3 ($= 3^1$ sur le niveau 2), 9 ($= 3^2$ sur le niveau 3) et 27 ($= 3^3$ sur le niveau 4) états.

La complexité temporelle (resp. spatiale) est donc mesurée par le nombre d'états générés et explorés (resp. nombre maximal d'états conservés en mémoire).

L'algorithme 1 est l'algorithme de recherche le plus général.

Algorithme 1 : Algorithme de recherche général
Données :
e_0 : état initial,
`FilsEtat` : produit les fils d'un état,
`TestBut` : prédicat qui teste si un état est un état-but,
`Classe` : range les fils d'un état dans la liste EnAttente
Variables :
EnAttente, Vus
e : état courant , *trouve* : un booléen
début
EnAttente ← (e_0)
Vus ← ()
trouve ← FALSE
tant que *EnAttente non vide et non trouve* **faire**
e ← choisirEtEnlever(EnAttente)
Vus ← ajouter(e, Vus)
si `TestBut`(e) **alors**
trouve ← TRUE
sinon
EnAttente ← `Classe`(`FilsEtat`(e), EnAttente, Vus)
si *non trouve* **alors**
print(ECHEC)
sinon
retourner e

Notons que différentes optimisations sont possibles concernant la fonction `Classe` :
— La fonction `Classe` n'ajoute qu'une seule occurrence de chaque état

— L'utilisation de la liste Vus permet d'éviter de développer un état qui a déjà été développé (et qui n'apparaît donc plus dans la liste EnAttente). Pour intégrer cette optimisation, la fonction `Classe` n'ajoute à la liste EnAttente que les fils de l'état développé qui ne sont pas déjà dans Vus. Attention, cette optimisation ne peut fonctionner que si on est certain de ne jamais retomber sur un état déjà developpé en passant par un meilleur chemin.

7.1.1 Exemples de stratégies non informées

Une stratégie définit le choix de l'état à considérer pour poursuivre la recherche. Une stratégie est dite non informée si le choix de l'état à développer est basé sur des critères syntaxiques.

Stratégie Largeur d'abord La fonction `Classe` ajoute les états à la fin de la liste EnAttente (EnAttente est donc une file).

Stratégie Profondeur d'abord [1] La fonction `Classe` ajoute les états en tête de la liste EnAttente (EnAttente est donc une pile).

Profondeur bornée Profondeur d'abord avec une limite (l) sur la profondeur (les états de profondeur $\geq l$ ne sont pas développés).

Profondeur itérative Profondeur bornée en essayant des profondeurs successives (1, 2, 3, ...).

Les propriétés de ces différents algorithmes sont :
• Un algorithme de recherche en largeur d'abord est complet (pourvu que b soit fini) et le nombre d'états créés [2] est en $O(b^{d+1})$. La complexité temporelle est fonction du nombre d'états développés et est donc en $O(b^d)$. La complexité spatiale est fonction du nombre d'états créés et est donc en $O(b^{d+1})$.
• Un algorithme de recherche en profondeur d'abord est complet si on intègre les optimisations de la fonction `Classe` (on évite les états répétés) [3]. Le nombre d'états créés est en $O(b^m)$ (la complexité temporelle est en $O(b^m)$) ; le nombre d'états à stocker est en $O(bm)$ si on ne tient compte que de la liste EnAttente (ce qui donne une complexité spatiale linéaire), mais est en $O(b^m)$ si on tient compte aussi de la liste Vus.
• Un algorithme de recherche en profondeur bornée est complet si $l > d$ (rappelons que d est la profondeur d'un état-but le moins profond). La complexité temporelle est en $O(b^l)$; le nombre d'états à stocker est en $O(bl)$.
• Un algorithme de recherche en profondeur itérative combine les avantages de Largeur d'abord (complétude si b est fini) et de Profondeur d'abord (complexité spatiale linéaire).

La figure 7.1 page suivante illustre la différence dans l'ordre du développement sur un même espace de recherche.

7.2 Recherche informée et heuristiques

La stratégie peut utiliser une heuristique, qui dépend du problème. On appelle heuristique tout procédé nous guidant dans la recherche de(s) solution(s). Une heuristique peut par exemple utiliser :

1. Les algorithmes obtenus avec les stratégies Largeur d'abord et Profondeur d'abord ont été vus dans le cours de Graphes.
2. Dans le pire des cas, les états développés sont tous ceux situés à une profondeur $\leq$ à celle de l'état-but le plus proche de la racine (donc $\leq d$) ; et les états créés sont les états développés plus quasiment tous ceux à la profondeur suivant directement celle de l'état-but (donc $d + 1$). D'autre part, on utilise le résultat mathématique suivant : $1 + x + x^2 + \ldots + x^n = \frac{1-x^{n+1}}{1-x}$.
3. sinon la complétude n'est pas garantie dans le cas d'un espace de recherche infini comme dans l'exemple 2 page 4 en section 7.2.1 page 38.

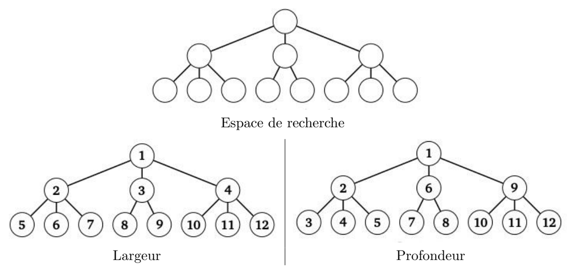
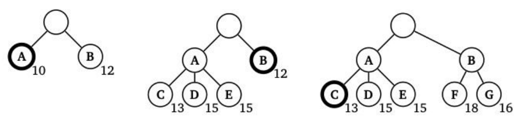


FIGURE 7.1 – Recherche en largeur et en profondeur

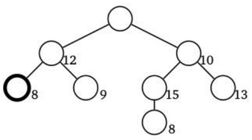
- un classement des opérateurs applicables à un état (du moins prometteur vers le plus prometteur). Cette information permettra de choisir parmi les fils d'un état ;
- une fonction d'évaluation des états (estimant par exemple la distance entre un état et l'état-but le plus proche). Cette information permettra de choisir parmi tous les états de EnAttente. Elle permettra aussi de réviser les choix, comme illustré sur la figure ci-dessous.



On utilise une fonction d'évaluation des états notée f qui permet de comparer tous les états de la liste EnAttente, quelle que soit leur position dans l'arbre développé. L'état à développer est choisi selon la valeur donnée par la fonction d'évaluation. L'évaluation d'un état peut mesurer par exemple :

— la difficulté de la résolution du sous-problème représenté par l'état,
— la qualité (ou inversement le coût) d'un chemin solution complet partant de l'état initial et passant par l'état.

Dans le cas où l'évaluation mesure un coût associé à l'état, on développe d'abord l'état dont l'évaluation est la plus faible.



Exemple du taquin : la fonction d'évaluation compte le nombre de jetons mal placés.

L'algorithme 2 est un algorithme de recherche qui utilise une stratégie meilleur d'abord. La fonction `Classe` utilise maintenant une fonction d'évaluation d'état ; elle est donc renommée `ClasseSelonEval` et ajoute maintenant à la liste EnAttente les fils de l'état développé qui ne sont pas dans Vus, ou qui améliorent un état de Vus, en maintenant la liste EnAttente triée par évaluation décroissante (les meilleurs états sont en tête).

Algorithme 2 : Stratégie meilleur d'abord

Données :
e_0 : état initial,
`FilsEtat` : produit les fils d'un état,
`TestBut` : prédicat qui teste si un état est un état-but,
`ClasseSelonEval` : range les fils d'un état dans la liste EnAttente selon l'évaluation
Variables :
EnAttente, Vus
e : état courant , *trouve* : un booléen
début
EnAttente ← (e_0)
Vus ← ()
trouve ← FALSE
tant que *EnAttente non vide et non trouve* **faire**
e ← EnleverTete(EnAttente)
Vus ← ajouter(e, Vus)
si `TestBut`*(e)* **alors**
trouve ← TRUE
sinon
EnAttente ← `ClasseSelonEval`(`FilsEtat`(e), EnAttente, Vus)
si *non trouve* **alors**
print(ECHEC)
sinon
retourner e

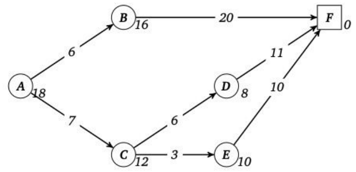
La plupart des stratégies meilleur d'abord intègrent comme composante de la fonction f une fonction heuristique, généralement notée h, dont la valeur dépend uniquement de l'état, et non du déroulement de la recherche.
Nous allons maintenant considérer des cas particuliers de stratégie meilleur d'abord avec une fonction heuristique h estimant le coût minimal d'un chemin menant de l'état courant à un état-but. Ces stratégies nous permettront aussi d'aborder le problème de la recherche d'une solution optimale, dans le cas où l'on dispose d'un critère de comparaison de solutions.
Nous supposons donc maintenant que chaque opérateur a un coût. Le coût d'une solution est donc

le coût du chemin associé.

7.2.1 Stratégie Meilleur d'abord Gloutonne

La fonction d'évaluation f se réduit à la composante heuristique h, qui estime le coût minimal d'un chemin menant de l'état courant à un état-but.
Complétude : Oui, si l'espace de recherche est fini avec vérification des états répétés (on intègre les optimisations de la fonction `Classe`)
Complexité temporelle et spatiale en $O(b^m)$.
Ce n'est pas un algorithme admissible (voir exemple 13).

Exemple 13 *Considérons une carte indiquant le sens et le coût des déplacements entre différents points. Le problème est de trouver un chemin de coût minimal menant de A à F. La valeur de l'heuristique h est indiquée sur chaque point de la carte.*



Dans cet exemple, les sommets développés sont successivement A, C et D. Le chemin trouvé sera donc ACDF de coût 24 alors qu'il existe un autre chemin qui avait un meilleur coût (ACEF de coût 20). Cela montre bien que cet algorithme n'est pas admissible.

7.2.2 Stratégie A*

Un cas particulier très connu de stratégie meilleur d'abord est la stratégie A*, dans laquelle la fonction f appliquée à un état donne une estimation du coût du chemin optimal menant de l'état initial à un état but en passant par l'état courant (coût de la solution optimale passant par l'état courant).

Pour chaque état e, on définit :
— $g^*(e)$ le coût du chemin optimal menant de l'état initial à l'état e,
— $h^*(e)$ le coût d'un chemin optimal menant de l'état courant à un état-but,
— $f^*(e) = g^*(e) + h^*(e)$.

On obtient une estimation f de f^* par $f = g+h$, où g est une estimation de g^* et h une estimation de h^*. Un choix naturel pour $g(e)$ est le coût du chemin menant de l'état initial à l'état courant e. On a donc toujours $g(e) \geq g^*(e)$. L'estimation h de h^* est indépendante du déroulement de la recherche et traduit une connaissance heuristique liée au problème lui-même (par exemple dans le jeu du taquin le nombre de jetons mal placés).
Dans le cas particulier où $h = 0$, (plus généralement h constante), on obtient l'algorithme dit de "Coût Uniforme" (noter que sans l'ensemble Vus on retrouve la version de base de l'algorithme de Dijkstra en théorie des graphes, voir section 7.3 page 40). Si de plus tous les coûts d'arcs valent 1, on retrouve une recherche en largeur d'abord.

Caractéristiques d'une heuristique

— h est une heuristique *monotone* ssi pour tout état u et pour tout état v fils de u, on a $h(u) \leq h(v)+$ coût (u, v)
— h est une heuristique *coïncidente* ssi pour tout état-but e, $h(e) = 0$.
— h est une heuristique *minorante* (ou optimiste) ssi pour tout état e, $h(e) \leq h^*(e)$

Propriété 1 *Une heuristique coïncidente et monotone est minorante.*

Propriétés d'un algorithme utilisant la stratégie A*
Considérons un problème pour lequel *l'espace de recherche contient un état but*, le nombre de successeurs d'un état est fini et il existe un minorant strictement positif de l'ensemble des coûts des arcs. Sous ces conditions :

1. Complétude : tout algorithme de type A* se termine en découvrant un chemin menant de l'état initial à un état-but. En conséquence, un algorithme de Coût Uniforme trouve lui-aussi un chemin de coût minimal.
2. Admissibilité : si h est minorante alors on obtient un algorithme A* admissible. (C'est le cas pour h = nombre de jetons mal placés dans le taquin). Par abus de langage, on dira alors que h est *admissible*.
3. Si h est monotone, un état rangé dans la liste Vus ne sera jamais amélioré (donc lorsque le fils d'un état est produit, s'il est présent dans la liste Vus, on ne le rajoute pas à la liste EnAttente).
4. Complexité : dans le pire cas, A* doit mémoriser tous les nœuds et la complexité en temps est en 2^N où N est la taille de l'espace de recherche. Mais la performance des algorithmes de recherche heuristique dépend de la qualité de la fonction heuristique.

Exemple 13 page ci-contre (cont'd) *L'heuristique h est minorante. On vérifie que la solution trouvée par A* est de coût minimal.*

Preuve de l'admissibilité d'une heuristique minorante
On considère le cas d'une heuristique h à valeurs positives, et d'un graphe G admettant au moins un chemin fini de l'état initial à un état but.

Lemme 1 *A tout moment avant que A* ne se termine, il existe un état e_i dans EnAttente qui appartient à un chemin optimal de l'état initial à un état but et tel que $g(e_i) = g^*(e_i)$.*

> **Preuve :** Soit $(e_0, e_1, \ldots, e_n)$ un chemin optimal issu de e_0 (état initial). Au départ, e_0 se trouve dans EnAttente. Lors du développement de e_0, c'est e_1 qui entre dans EnAttente. Soit e_i le premier état de ce chemin qui se trouve dans EnAttente (*i.e.* l'état de EnAttente le moins profond de ce chemin). Puisque tous les ancêtres de e_i sont dans Vus, le sous-chemin $(e_0, e_1, \ldots, e_i)$ est connu. Comme c'est un sous-chemin d'un chemin optimal, il est optimal [4] pour arriver en e_i donc $g(e_i) = g^*(e_i)$.

Lemme 2 *Si l'heuristique h est minorante alors à la fin de chaque itération de l'algorithme, l'état e en-tête de EnAttente est tel que $f(e) \leq f^*(e_0)$.*

> **Preuve :** Par construction, puisque e est en-tête de EnAttente, $f(e) = min\{f(o), o \in$ *EnAttente*$\}$. D'après le lemme 1, à tout moment il existe dans EnAttente un état e_i sur un chemin optimal tel que $g(e_i) = g^*(e_i)$. Comme h est minorante, $h(e_i) \leq h^*(e_i)$. On a donc $f(e_i) \leq f^*(e_i) = f^*(e_0)$. Comme $f(e) \leq f(e_i)$ on a donc $f(e) \leq f^*(e_0)$.

Propriété 2 *Si l'heuristique h est minorante, l'algorithme A* est admissible.*

> **Preuve :** L'algorithme A* se termine. Il finit donc par rencontrer un état e en tête de EnAttente qui est un état but. D'après le lemme 2, $f(e) \leq f^*(e_0)$. Cet état est donc le dernier état d'un chemin optimal.

4. C'est le principe d'optimalité de Bellman.

7.3 Comparaison avec des algorithmes classiques en théorie des graphes

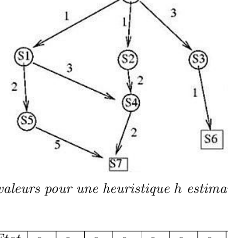
Considérons l'algorithme de Moore-Djikstra présenté en cours de Graphes l'an dernier [5] :
Etant donné un sommet source e_0 et un sommet e différent de e_0, l'algorithme de Moore-Djikstra fournit un chemin optimal (c'est-à-dire de coût minimal) de e_0 à e. La stratégie utilisée est celle du coût uniforme. Elle ne tient donc pas compte de la distance au but pour choisir le prochain état à étudier.
D'autre part, le graphe orienté pondéré positivement est explicitement disponible au départ.
Alors que les algorithmes vus dans la section précédente pilotent l'exploration progressive du graphe des états sans qu'il soit utile de le construire en totalité.

7.4 Exercices

Énoncé 21 *Faire le parcours en profondeur et en largeur sur le problème Die Hard.*

Énoncé 22 *Proposez une heuristique pour le problème de satisfaction du sac à dos suivant :*
— *poids :* $\{4, 3, 8, 4, 4, 9\}$
— *poids max :* 23

Énoncé 23 *Soit le graphe d'état valué représenté sur la figure suivante. Chaque état est noté s_i, i allant de 0 à 7 ; s_0 est l'état initial. Les états solutions sont entourés d'un rectangle.*



Le tableau suivant donne les valeurs pour une heuristique h estimant la distance à la solution pour chaque état.

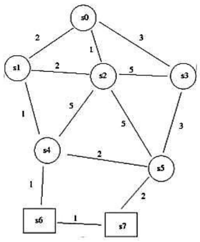
Etat	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
Val	4	4	3	3	5	2	0	0

1. *Développer la méthode d'exploration dite du "coût uniforme" jusqu'à une solution. Commentez la solution trouvée.*
2. *Donnez l'état de la liste des noeuds en attente pendant le déroulement de l'algorithme A*, en utilisant l'heuristique h. Vous préciserez comment sont effectués les changements sur cette liste. Commentez le résultat. Que peut-on dire de l'heuristique h ?*

Énoncé 24 *On considère le graphe d'états représenté sur la figure ci-après. Le graphe est non-orienté, les valeurs sur les arcs indiquent le coût de passage d'un état à un autre. On considère*

5. Pour une comparaison plus détaillée, voir l'article de F. Rossi "L'algorithme A* ou comment calculer rapidement un chemin optimal" GNU/Linux Magazine France, volume 54, pages 32-41, Octobre 2003.

que l'on dispose d'une fonction qui reconnaît les états solutions, indiqués par un rectangle sur la figure.



Le tableau suivant donne les valeurs pour deux heuristiques h_1 et h_2 estimant la distance à la solution pour chaque état.

Etat	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
h_1	0	6	8	2	1	1	0	0
h_2	0	3	8	3	1	1	0	0

Pour chacun des algorithmes suivants, vous détaillerez l'évolution de la recherche jusqu'à une solution en partant de s_0 (donnez l'état courant, le contenu de la liste des ouverts, de la liste des fermés). Vous donnerez aussi le nombre total d'états générés, et le nombre total d'états développés.

1. *A* avec l'heuristique h_1,*
2. *A* avec l'heuristique h_2.*

L'heuristique h_1 est-elle optimiste ? Et h_2 ?

Énoncé 25 *On dispose de la matrice de distance suivante entre 4 villes :*

	a	*b*	*c*	*d*
a	—	2	5	7
b	2	—	8	3
c	5	8	—	1
d	7	3	1	—

1. *Appliquer l'algorithme du coût uniforme pour trouver le plus court chemin allant de la ville a à la ville d.*
2. *Si on voulait appliquer le A*, que pourrait-on utiliser comme heuristique ?*

Énoncé 26 *On réutilise le graphe des villes donné à l'énoncé 25.*
Le but du voyageur de commerce est de visiter chacune de ces villes (sans jamais y passer deux fois).

1. *Donner une formalisation du problème du voyageur de commerce. Comment définit-on un état final ?*
2. *Quelle est la taille de l'espace des états ? Peut-on le réduire ?*
3. *Développer la méthode d'exploration dite du "coût uniforme" jusqu'à une solution.*

4. *Utiliser l'heuristique $h(e)$ =(nb d'arêtes manquantes dans e)×(coût minimal d'arête dans le graphe) pour appliquer l'algo A*.*

Bilan :
• Une méthode complète est un parcours de l'espace d'état jusqu'à atteindre un état but ou jusqu'à avoir tout exploré.
• Très coûteux en temps !
• 2 possibilités :
  ○ non informée : pas de stratégie pour choisir les états (largeur d'abord / profondeur d'abord),
  ○ informée : utilisation d'une stratégie pour arriver plus vite sur un état but (meilleur d'abord / coût uniforme / A*).
• Distinction à faire entre "état but" (pb de satisfaction) et "meilleur état but" (pb d'optimisation).
• Dans le cas d'un problème d'optimisation, pas de stratégie ou stratégie admissible : garantie de trouver la meilleure solution.
• Heuristique : estimation du coût restant à parcourir avant atteindre un état but.
• Propriétés d'une heuristique : monotone, coïncidente, minorante, admissible.

Chapitre 8

Méthodes incomplètes

Toutes les méthodes complètes présentées précédemment ont l'inconvénient majeur d'être extrêmement consommatrices en temps. En effet :

- Les méthodes complètes/exactes explorent de façon systématique l'espace de recherche. Cela ne marche donc pas dans de nombreux problèmes à cause de l'explosion combinatoire du nb d'états possibles.
- Ces méthodes sont généralement à base de recherche arborescente. On est donc contraint par l'ordre des noeuds dans l'arbre. Cela implique une impossibilité de compromis qualité/temps ($\neq$ algo "anytime" [1]).

Or il y a souvent des cas où on peut se contenter d'une solution approchée, pourvu qu'elle soit suffisamment "bonne".
Pour trouver cette solution approchée, on a deux types de stratégie :

- garder une méthode complète et stopper après un temps déterminé (mais suivant les problèmes, cela n'est pas toujours possible)
- plonger dans l'arbre de recherche avec des heuristiques sans jamais revenir en arrière (mais : comment être sûr qu'on va trouver vite une bonne solution ?)

La seconde possibilité correspond à la notion de méthodes "incomplètes", dites aussi "méthodes approchées" (ou méthodes "heuristiques" [2]). Ces méthodes ne garantissent pas d'obtenir la solution optimale mais sont en général suffisamment "efficaces" pour être exploitables : le rapport qualité de la solution trouvée/temps mis pour la trouver est raisonnablement bon.

Il existe beaucoup de familles de méthodes différentes. Nous allons aborder ici uniquement la famille des **méthodes locales** dont le principe est le suivant :

1. Travailler avec un nouvel espace : celui des solutions (il est différent de l'espace des états mais ils peuvent avoir des sommets en commun).
2. Partir d'une solution sinon approchée, du moins potentiellement bonne et essayer de l'améliorer itérativement. Pour améliorer une solution on ne fait que de légers changements (on parle de changement local, ou de solution voisine).
3. Relancer la méthode plusieurs fois en changeant le point de départ pour avoir plus de couverture.
4. Tout problème est considéré comme un problème d'optimisation (même les problèmes de satisfaction : le coût à optimiser est alors le nombre de contraintes insatisfaites).

1. Un algorithme anytime est un algorithme capable de donner une solution valide à un problème même s'il est interrompu avant d'avoir terminé. L'algorithme trouve de meilleures solutions au fur et à mesure de son exécution.
2. Attention à ne pas mélanger avec la notion d'heuristique présentée pour l'algorithme A*. Ici le mot "heuristique" s'entend comme "exploitant une stratégie pour aller plus vite" et pas comme une métrique associée à chaque état et estimant le coût restant avant d'atteindre la solution. Ceci étant, le fait que le même mot soit utilisé montre bien que cette métrique sert aussi de stratégie pour aller plus vite.

5. Ici la notion de complétude est liée à la découverte de la meilleure solution (et plus à la découverte d'une solution ! Cela correspond donc à la notion d'admissibilité des méthodes complètes).

Précisons le vocabulaire utilisé ici :

Définition 14

Solution : *c'est un état-but du problème tel que défini pour les méthodes complètes (dans le sens où il permet d'affecter une valeur à toutes les variables du pb).*

Fonction d'évaluation `eval` **:** *c'est une fonction qui évalue un état solution.*

Solution optimale : *c'est une solution de coût minimal (resp. maximal) selon* `eval`.

Mouvement : *c'est une opération élémentaire permettant de passer d'une solution à une solution voisine.*

Voisinage d'une solution : *c'est l'ensemble des solutions voisines, c'est-à-dire l'ensemble des solutions accessibles par un mouvement (et un seul).*

On a donc l'espace des solutions qui pourrait être représenté sous forme graphique avec les solutions servant de sommets et les mouvements permettant de définir les arcs entre deux solutions voisines.

Un algorithme de recherche locale typique pour le cas d'une minimisation est l'algorithme 3.

Algorithme 3 : Algorithme de recherche locale (pour minimisation) "sans reprise"

Données :
  max_mouvements : le nb max de mouvements à faire,
  `nouvelle_solution` : la fonction générant une solution (aléatoirement ou non),
  `eval` : la fonction d'évaluation,
  `choisir_voisin` : la fonction de choix du voisin à exploiter

Variables :
  E : solution courante,
  E* : la meilleure solution atteinte,
  E' : le voisin choisi,
  m : le mouvement courant

début
  E ← `nouvelle_solution`()
  E* ← E
  pour *m=1 à max_mouvements* **faire**
    E' ← `choisir_voisin`(E)
    si `eval`*(E')* < `eval`*(E)* **alors**
      // on améliore E
      E ← E'
  si `eval`*(E)*<`eval`*(E*)* **alors**
    // on améliore E*
    E* ← E
  retourner E*,`eval`(E*)

On peut ensuite améliorer cet algorithme en le répétant plusieurs fois avec une solution initiale différente dès qu'on ne peut plus améliorer (ce sera la version "avec reprise" illustrée dans l'algorithme 4). Cette nouvelle version est elle-aussi incomplète mais, à défaut, elle permettra peut-être de trouver un meilleur optimum local.

Algorithme 4 : Algorithme recherche locale (pour minimisation) "avec reprise" si pas d'amélioration

```
Données :
    max_reprises : le nb max de reprises à faire,
    nouvelle_solution : la fonction générant une solution (aléatoirement ou non),
    eval : la fonction d'évaluation,
    choisir_voisin : la fonction de choix du voisin à exploiter
Variables :
    E : solution courante,
    E* : la meilleure solution atteinte,
    E' : le voisin choisi,
    r : la reprise courante
début
  E ← nouvelle_solution()
  E* ← E
  pour r=1 à max_reprises faire
    E' ← choisir_voisin(E)
    tant que eval(E') < eval(E) faire
      // on améliore E
      E ← E'
      E' ← choisir_voisin(E)
    si eval(E)<eval(E*) alors
      // on améliore E*
      E* ← E
    E ← nouvelle_solution()
  retourner E*,eval(E*)
```

De manière générale, ce type d'algo se comporte de la manière suivante :

1. une majorité de mouvements améliore la solution courante,
2. puis le nombre d'améliorations devient de plus en plus faible,
3. il n'y a plus d'améliorations : on est dans un optimum, qui peut être local,
4. à chaque reprise, on espère explorer une autre partie de l'espace des solutions mais on n'améliore pas forcément l'optimum local déjà trouvé.

Avec ce type de méthode, les questions à se poser sont les suivantes :
- quand faut-il s'arrêter ? Plus précisément :
  - pour la reprise en cours : nb de mouvements max ? comment détecter un optimum local ?
  - pour la recherche elle-même : le nb max de reprises ? coût de la solution convenable ? limite de temps atteinte ?

- faut-il être opportuniste ou gourmand [3] ? Plus précisément :
  - si aucune recherche d'amélioration : comment trouver les bonnes solutions ?
  - si recherche d'amélioration, comment éviter les optimums locaux ?
- comment ajuster les paramètres nombre de reprises/nombre de mouvements ?
- comment comparer les performances de deux méthodes différentes ? (qualité de la solution vs. temps consommé)

Dans la suite, nous allons successivement illustrer ces idées d'algorithme sur différentes méthodes : le *Hill-climbing* (ou escalade), le *Steepest Hill-climbing* (ou escalade par la plus grande pente), le *Tabou*.

8.1 Les stratégies Hill-Climbing et Steepest Hill-Climbing

Le Hill-Climbing est aussi appelé recherche par gradient [4]. Ici on a :
- `choisir_voisin` : choix aléatoire dans le voisinage courant,
- mise à jour de E : seulement s'il y a une amélioration.

C'est donc une méthode opportuniste (ne cherche pas à trouver le meilleur voisin mais n'utilise un voisin que s'il est meilleur que la solution courante).
Il s'agit donc d'une stratégie profondeur d'abord combinée avec le meilleur des fils selon une fonction heuristique. Elle est principalement utilisée pour la recherche d'une solution optimale.
Cette stratégie ne maintient pas d'arbre de recherche (on ne revient jamais en arrière) et ne regarde pas très loin. Le risque est d'être bloqué sur un optimum local, même en utilisant la version "avec reprise" (algorithme 4 page précédente).

Le Steepest Hill-Climbing est une version plus perfectionnée du Hill-Climbing qui consiste à prendre "la plus grande pente". Ici on a :
- `choisir_voisin` : après avoir déterminé l'ensemble des meilleures solutions voisines de la solution courante (exceptée celle-ci), on en choisit une aléatoirement,
- mise à jour de E : si amélioration (donc arrêt sur optimum local).

C'est donc un algorithme glouton (*greedy* : choix de l'optimum local à chaque étape – choix du voisin et mise à jour de E).
Comme pour le Hill-Climbing, cette stratégie ne maintient pas d'arbre de recherche (on ne revient jamais en arrière) et ne regarde pas très loin. Elle regarde juste "un peu mieux autour". Le risque est, là-aussi, d'être bloqué sur un optimum local, même en utilisant la version "avec reprise" (algorithme 4 page précédente).

La différence de comportement entre les deux algos est illustrée sur la figure 8.1 page ci-contre.

8.2 Stratégie "Tabou"

C'est aussi une méthode locale basée sur une recherche par gradient avec les spécificités suivantes :
- choix dans le voisinage,
- possibilité de détérioration de la solution courante,

3. Un algorithme glouton (*greedy algorithm* en anglais, parfois appelé aussi algorithme gourmand) est un algorithme qui suit le principe de faire, étape par étape, un choix d'optimum local. Dans certains cas, cette approche permet d'arriver à un optimum global, mais dans le cas général c'est une simple heuristique. Cette notion s'oppose à celle d'un algorithme opportuniste qui ne cherche pas à satisfaire un optimum local mais qui n'utilise un voisin que s'il est meilleur que la solution courante.

4. On utilise aussi les termes "descente en gradient" ou "montée en gradient", suivant le sens de l'optimisation (min ou max) recherché.

en montant le plus tôt possible — en suivant la pente la plus forte

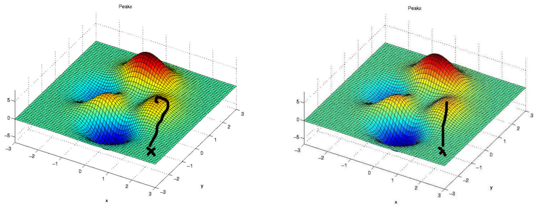


Figure 8.1 – Différence entre Hill Climbing et Steepest Hill-Climbing avec l'algorithme "sans reprise"

- pour améliorer la recherche, on mémorise les k dernières solutions courantes et on interdit le retour sur une de ces solutions,
- on autorisera en plus toujours un mouvement conduisant à une meilleure solution que la meilleure obtenue auparavant.

En pratique, au lieu de mémoriser les k dernières solutions courantes (parfois trop coûteux en temps et mémoire), on mémorise les k derniers mouvements (plus restrictif mais peu coûteux en temps).
L'algorithme dans le cas d'une minimisation est l'algorithme 5 page suivante.

8.3 Conclusion sur les méthodes locales

Les méthodes incomplètes correspondent à un ensemble de méthodes empiriques variées, parfois trop, car l'absence de modélisation les rend difficiles à adapter d'un problème à un autre (cela fait un peu bricolage). Par contre, elles sont à peu près indifférentes à la taille du problème, et donnent de bons résultats (approchés) à condition de bien les paramétrer. Elles peuvent donc fournir un bon complément aux méthodes complètes.
Quelques pistes d'amélioration : Pour éviter d'être coincé dans un optimum local, on peut ajouter du bruit : une part de mouvements aléatoires pendant une recherche classique en suivant le principe :

> avec une proba p, faire un mouvement aléatoire,
> avec une proba $1-p$, suivre la méthode originale.

On peut jouer aussi au niveau de l'acceptabilité d'un voisin pour la mise à jour de la solution courante : s'il y a une dégradation, l'accepter avec une probabilité p.
Reste le problème : comment régler p ? On a toujours des compromis à faire entre exploration globale/locale.

8.4 Exercices

Énoncé 27 *Définir les fonctions de création d'une solution, d'évaluation et de voisinage pour le problème du sac à dos.*

Énoncé 28 *Soit l'espace des solutions donné sur la figure 8.2.*

Algorithme 5 : Algorithme Tabou (pour minimisation)

```
Données :
    condition_de_fin : cela peut être une limite en temps, en itération, ou une
        non amélioration de la solution,
    nouvelle_solution : la fonction générant une solution (aléatoirement ou non),
    eval : la fonction d'évaluation,
    voisinage : l'ensemble des voisins d'une solution
Variables :
    E : solution courante,
    E* : la meilleure solution atteinte,
    V : l'ensemble des voisins de la solution courante qui ne sont pas tabous,
    E' : le voisin choisi,
    T : la liste des solutions tabous
début
  E ← nouvelle_solution()
  E* ← E
  T ← ∅
  tant que non(condition_de_fin) faire
    V ← voisinage(E) − T
    E' ← min pour eval sur V
    si eval(E') ≥ eval(E) alors
      // on ne peut plus améliorer
      T ← T ∪ {E}
    si eval(E*) > eval(E') alors
      // on améliore E*
      E* ← E'
    // dans tous les cas, on repart du voisin
    E ← E'
  retourner E*, eval(E*)
```

1. *Décrivez le fonctionnement du Steepest Hill-Climbing sans reprise sur cet espace avec les hypothèses suivantes :*
   - *on cherche la solution avec la valeur maximum,*
   - *la fonction* `nouvelle_solution` *renvoie* s_0.
2. *Décrivez le fonctionnement du Tabou sans reprise sur cet espace avec les hypothèses suivantes :*
   - *on cherche la solution avec la valeur maximum,*
   - *la fonction* `nouvelle_solution` *renvoie* s_0,
   - *la liste Tabou est illimitée,*
   - *la condition de fin correspond au remplissage de la liste Tabou avec au moins 50% des solutions de l'espace.*

Énoncé 29 *Appliquer le steepest hill climbing au problème du sac à dos avec :*
- *poids :* $\{4, 3, 8, 4, 4, 9\}$
- *valeurs :* $\{7, 2, 32, 9, 2, 8\}$
- *poids max :* 23

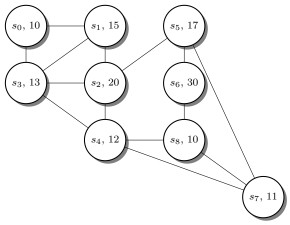


FIGURE 8.2 – Espace des solutions pour l'énoncé 28

Bilan :
• Méthode incomplète : pas de parcours exhaustif de l'espace des états, ni de celui des solutions.
• Pas de garantie d'obtenir la meilleure solution mais plus rapide qu'une méthode complète.
• Plusieurs possibilités : ici uniquement par recherche locale et donc raisonnement sur l'espace des solutions et pas l'espace des états.
• Le principe : on part d'une solution et on cherche à l'améliorer en étudiant ses voisines.
• Donne des solutions approchées (parfois des optimums locaux).
• Deux exemples : les Hill-Climbing et le Tabou.

Chapitre 9

CSP : représentation et résolution

Il s'agit là-aussi d'une représentation à partir de graphes mais avec des graphes plus spécifiques car représentant exclusivement des contraintes. Du coup, les algorithmes de résolution seront des variantes spécialisées des algorithmes de parcours vus précédemment.

9.1 Definition

Rappelons que dans les problèmes de recherche "classiques", on a les éléments suivants :
— Un *état* est une "boite noire" et
— cette "boite noire" correspond à n'importe quelle structure de données contenant a minima un *test pour le but*, une fonction d'*évaluation* et une fonction *successeur*.

Par opposition, dans les CSP, on a :
— Un *état* est défini par un ensemble de variables V_i, dont les valeurs appartiennent au domaine D_i ;
— le *test pour le but*, la fonction d'*évaluation* et la fonction *successeur* sont définies à l'aide d'un ensemble de contraintes qui spécifient les combinaisons autorisées pour les valeurs sur des sous-ensembles de variables, sachant qu'on cherche au final à affecter une valeur à chaque variable en respectant les contraintes.

Il existe une grande variété de CSPs :
— suivant le type de variable : discrète ou continue,
— suivant le type de domaine : fini ou infini,
— suivant le type de contrainte : linéaire ou pas, unaire, binaire ou n-aire, représentant des contraintes strictes ou des préférences, ...

Dans ce cours, on va se contenter de CSP manipulant des variables discrètes à domaines finis et des contraintes strictes.

Définition 15 (CSP) *Un CSP est un triplet* (V, D, C) *avec :*
— *V l'ensemble des variables* $\{V_1, V_2, V_3, \ldots\}$,
— *D l'ensemble des domaines de valeurs, au plus un par variable,* $\{D_1, D_2, D_3, \ldots\}$ *(les valeurs de* V_i *étant prises dans le domaine* D_i*),*
— *C l'ensemble des contraintes entre variables qui définissent des tuples possibles de valeurs sur les domaines, chaque contrainte étant notée* $C_{ij\ldots} = \{(x_i, y_j, \ldots) \in D_i \times D_j \times \ldots\}$

Quand on modélise un problème sous la forme d'un CSP, la difficulté principale est de choisir les bonnes variables car cela influe fortement sur l'expression des contraintes et donc sur l'efficacité de résolution.

9.2 Algorithmes

L'algorithme de recherche standard correspond à une recherche incrémentale :
— Les états sont définis par les valeurs des variables déjà affectées.
— Etat initial : Un ensemble d'affectations vide $\varnothing$.
— Fonction successeur : attribuer une valeur à une variable non encore affectée, de façon cohérente (par rapport aux contraintes) à l'affectation actuelle.
— Test du but : l'affectation courante est complète.

Cet algorithme de recherche marche pour tous les CSPs. Chaque solution apparaît à une profondeur de n s'il y a n variables ; cela revient à utiliser le principe de la recherche en profondeur d'abord.

Il existe beaucoup de variantes de cet algorithme standard. Nous nous contenterons de voir ici la variante la plus simple, appelée *backtrack*, et une de ses améliorations (le *backtrack avec ordonnancement*).

De manière formelle, cela correspond à l'algorithme 6 qui utilise à son tour l'algorithme 7.

Algorithme 6 : Backtrack

Données :
  C : ensemble des contraintes
  D : matrice des domaines (le domaine de la variable i est la ligne i de D)
  n : nombre de variables
Variables locales :
  A : affectation courante de valeurs
  p : numéro de la variable courante
  D' : copie de D qui va évoluer au fur et à fur du déroulement de l'algo
  ok : résultat de la tentative d'affectation courante
début
  $p = 1$ // numéro première variable traitée (et aussi indice de profondeur)
  $A = \varnothing$ // affectation de valeurs vide au départ
  $D'[1]$ = copie de $D[1]$ // copie du domaine de la variable courante
  tant que $1 \leq p \leq n$ **faire**
    ok = SelectValeur(A,$D'[p]$,p,C) // essai d'affectation d'une valeur à la variable p
    si *not ok* **alors**
      $p = p - 1$ // cela n'a pas marché, on "backtrack"
      on enlève de A l'affectation concernant p // nettoyage de A
    sinon
      $p = p + 1$ // cela a marché, on passe à la variable suivante
      si $p \leq n$ **alors**
        $D'[p]$ = copie de $D[p]$ // en faisant une copie de son domaine
  si $p == 0$ **alors**
    print(ECHEC : CSP incohérent)
  sinon
    retourner A

Algorithme 7 : SelectValeur

Données :
- A : affectation courante de valeurs
- Dp : domaine courant de la variable p
- p : la variable pour laquelle on cherche une valeur
- C : ensemble des contraintes

Variables locales :
- v : valeur courante

// Attention : la donnée Dp est modifiée par l'appel de cet algorithme
// (toutes les valeurs choisies et incohérentes avec l'affectation courante ont été
// supprimées de Dp)
//
// Attention : la donnée A peut être modifiée par l'appel de cet algorithme
// si on trouve une valeur cohérente pour p

début
  tant que $Dp \neq \varnothing$ **faire**
    choisir $v \in Dp$ // choix d'une valeur pour p
    supprimer v de Dp // mise à jour de Dp
    si *$A \cup (p, v)$ est cohérente par rapport à C* **alors**
      $A = A \cup (p, v)$ // on rajoute l'affectation de p avec v à A
      retourner *VRAI* // on a trouvé une valeur pour p
  retourner *FAUX* // on n'a pas trouvé de valeur pour p, A reste inchangée

Une version améliorée de l'algorithme de backtrack consiste à ordonner au préalable les variables. Un critère d'ordonnancement classique est le degré de cette variable dans le graphe des contraintes. En effet, plus une variable est impliquée dans des contraintes et plus le choix de sa valeur risque d'impacter d'autres variables. Cela donne l'algorithme de backtrack avec ordonnancement (voir l'algorithme 8).

Algorithme 8 : Backtrack avec ordonnancement

Données :
- C : ensemble des contraintes
- D : matrice des domaines (le domaine de la variable i est la ligne i de D)
- n : nombre de variables

début
  ré-ordonner D en fonction du degré décroissant des variables dans le graphe des contraintes
  retourner *Backtrack(C,D,n)*

Notons que d'autres choix d'ordonnancement peuvent être faits (prise en compte de la taille des domaines par exemple). On peut aussi avoir des critères de choix de la valeur à assigner (la moins contraignante par exemple).

Et enfin, on peut aussi utiliser la propagation de contraintes pour anticiper l'impact d'une affectation. Cela consiste à propager le choix d'une valeur sur les domaines des variables non encore affectées. Cela peut se faire à chaque étape du backtrack. Cette variante est appelée le *forward checking*.

9.3 Exemples

Exemple 1 page 4 (cont'd) *Reprenons le problème de la coloration et codons-le maintenant sous la forme d'un CSP.*

Les variables correspondent aux sommets du graphe.
Les domaines des variables sont tous identiques et égaux à la liste des couleurs possibles.
Les contraintes sont données ici par les arêtes entre deux sommets.

Dans le cas du graphe $G = (X, E)$ avec $X = \{a, b, c\}$ et $E = \{(a, b), (b, c), (c, a)\}$, considérons d'abord le problème Pb_1 : Peut-on colorer G avec 3 couleurs ?

Le CSP correspondant est défini par :

- $V = X$
- $D = \{D_a, D_b, D_c\}$ *avec* $D_i = \{R, B, J\}$ *pour* $i = a, b, c$
- $C = \{C_{ab}, C_{bc}, C_{ca}\}$ *avec* $C_{ij} = \{(R, B), (R, J), (B, R), (B, J), (J, R), (J, B)\}$ *donnant la liste des doublons de valeurs autorisés pour les variables i et j quand $i \neq j$. Notons qu'on pourrait aussi choisir de ne mémoriser dans C_{ij} que les doublons interdits.*

Appliquons maintenant le backtrack pour résoudre le problème Pb_1.

$p = 1$ (1 étant le numéro de la variable a)
$A = \varnothing$ (initialisation de A)
$D'[1] = \{R, B, J\}$

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 1$ et $D'[1] = \{R, B, J\}$
  on essaye la valeur R pour 1 et $D'[1] = \{B, J\}$
  ce choix d'affectation est cohérent avec A (qui est vide pour l'instant)
  on met à jour A qui devient $\{(1, R)\}$
  on sort du SelectValeur en renvoyant VRAI
ok est VRAI donc $p = 2$ (2 étant le numéro de la variable b)
et $D'[2] = \{R, B, J\}$

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 2$ et $D'[2] = \{R, B, J\}$
  on essaye la valeur R pour 2 et $D'[2] = \{B, J\}$
  ce choix d'affectation est incohérent avec $A = \{(1, R)\}$
  on essaye la valeur B pour 2 et $D'[2] = \{J\}$
  ce choix d'affectation est cohérent avec $A = \{(1, R)\}$
  on met à jour A qui devient $\{(1, R), (2, B)\}$
  on sort du SelectValeur en renvoyant VRAI
ok est VRAI donc $p = 3$ (3 étant le numéro de la variable c)
et $D'[3] = \{R, B, J\}$

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 3$ et $D'[3] = \{R, B, J\}$
  on essaye la valeur R pour 3 et $D'[3] = \{B, J\}$
  ce choix d'affectation est incohérent avec $A = \{(1, R), (2, B)\}$
  on essaye la valeur B pour 3 et $D'[3] = \{J\}$
  ce choix d'affectation est incohérent avec $A = \{(1, R), (2, B)\}$
  on essaye la valeur J pour 3 et $D'[3] = \varnothing$
  ce choix d'affectation est cohérent avec $A = \{(1, R), (2, B)\}$
  on met à jour A qui devient $\{(1, R), (2, B), (3, J)\}$
  on sort du SelectValeur en renvoyant VRAI
ok est VRAI donc $p = 4$

Arrêt de la boucle puisque $p = 4$ et sortie en renvoyant A.

Remarquons ici deux choses : nous n'avons pas fait de backtrack et l'utilisation du backtrack avec ordonnancement n'aurait rien apporté ici puisque tous les sommets du graphe G ont le même degré.

Si on veut traiter maintenant le problème Pb_2, "Peut-on colorer G avec 2 couleurs ?", les domaines seront différents ($D_i = \{R, B\}$ pour $i = a, b, c$) et les contraintes modifiées en conséquence ($C_{ij} = \{(R, B), (B, R)\}$).

Et l'application du backtrack va donner la chose suivante.

$p = 1$ (1 étant le numéro de la variable a)
$A = \varnothing$ (initialisation de A)
$D'[1] = \{R, B\}$

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 1$ et $D'[1] = \{R, B\}$
  on essaye la valeur R pour 1 et $D'[1] = \{B\}$
  ce choix d'affectation est cohérent avec A (qui est vide pour l'instant)
  on met à jour A qui devient $\{(1, R)\}$
  on sort du SelectValeur en renvoyant VRAI
ok est VRAI donc $p = 2$ (2 étant le numéro de la variable b)
et $D'[2] = \{R, B\}$

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 2$ et $D'[2] = \{R, B\}$
  on essaye la valeur R pour 2 et $D'[2] = \{B\}$
  ce choix d'affectation est incohérent avec $A = \{(1, R)\}$
  on essaye la valeur B pour 2 et $D'[2] = \varnothing$
  ce choix d'affectation est cohérent avec $A = \{(1, R)\}$
  on met à jour A qui devient $\{(1, R), (2, B)\}$
  on sort du SelectValeur en renvoyant VRAI
ok est VRAI donc $p = 3$ (3 étant le numéro de la variable c)
et $D'[3] = \{R, B\}$

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 3$ et $D'[3] = \{R, B\}$
  on essaye la valeur R pour 3 et $D'[3] = \{B\}$
  ce choix d'affectation est incohérent avec $A = \{(1, R), (2, B)\}$
  on essaye la valeur B pour 3 et $D'[3] = \varnothing$
  ce choix d'affectation est incohérent avec $A = \{(1, R), (2, B)\}$
  $D'[3]$ est vide on sort du SelectValeur en renvoyant FAUX
ok est FAUX donc $p = 2$, $A = \{(1, R)\}$ $\Rightarrow$ ***backtrack***

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 2$ et $D'[2] = \varnothing$
  on sort du SelectValeur en renvoyant FAUX
ok est FAUX donc $p = 1$, $A = \varnothing$ $\Rightarrow$ ***backtrack***

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 1$ et $D'[1] = \{B\}$
  on essaye la valeur B pour 1 et $D'[1] = \varnothing$
  ce choix d'affectation est cohérent avec A (qui est vide pour l'instant)
  on met à jour A qui devient $\{(1, B)\}$
  on sort du SelectValeur en renvoyant VRAI
ok est VRAI donc $p = 2$ (2 étant le numéro de la variable b)
et $D'[2] = \{R, B\}$

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 2$ et $D'[2] = \{R, B\}$
  on essaye la valeur R pour 2 et $D'[2] = \{B\}$
  ce choix d'affectation est cohérent avec $A = \{(1, B)\}$
  on met à jour A qui devient $\{(1, B), (2, R)\}$
  on sort du SelectValeur en renvoyant VRAI
ok est VRAI donc $p = 3$ (3 étant le numéro de la variable c)
et $D'[3] = \{R, B\}$

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 3$ et $D'[3] = \{R, B\}$
  on essaye la valeur R pour 3 et $D'[3] = \{B\}$
  ce choix d'affectation est incohérent avec $A = \{(1, B), (2, R)\}$
  on essaye la valeur B pour 3 et $D'[3] = \varnothing$
  ce choix d'affectation est incohérent avec $A = \{(1, B), (2, R)\}$
  $D'[3]$ est vide, on sort du SelectValeur en renvoyant FAUX
ok est FAUX donc $p = 2$, $A = \{(1, B)\}$ $\Rightarrow$ ***backtrack***

p étant $\leq$ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 2$ et $D'[2] = \{B\}$
  on essaye la valeur B pour 2 et $D'[2] = \varnothing$
  ce choix d'affectation est incohérent avec $A = \{(1, B)\}$
  $D'[2]$ est vide, on sort du SelectValeur en renvoyant FAUX
ok est FAUX donc $p = 1$, $A = \varnothing$ $\Rightarrow$ ***backtrack***

p étant ≤ à 3 (nb de variables) on rentre dans la boucle
appel de SelectValeur avec $p = 1$ *et* $D'[1] = \emptyset$
on sort du SelectValeur en renvoyant FAUX
ok est FAUX donc $p = 0$, $A = \emptyset$ $\Rightarrow$ ***backtrack***

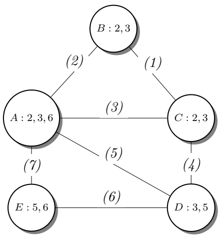
Arrêt de la boucle puisque $p < 1$
Sortie de l'algorithme avec le print ECHEC : CSP incohérent

9.4 Exercices

Énoncé 30 *Configuration de produits. Le but est de simuler la réalisation d'un produit complexe à partir de composants.*
Par exemple, pour un ordinateur il doit avoir un processeur (p), de la mémoire vive (m) et un disque dur (d) et on a le choix entre 3 types de p (p_1, p_2, p_3), *4 types de m* (m_1, m_2, m_3, m_4) *et 3 types de d* (d_1, d_2, d_3). *L'indice indique l'année de sortie (plus il est grand et plus c'est récent).*
Les contraintes sont les suivantes :
- p_1 *ne marche pas avec le composant* d_3.
- *Un processeur doit avoir une mémoire au moins aussi récente que lui (donc pas plus vieille).*
- *Le seul disque possible avec* $p_2 + m_2$ *est le disque* d_2.

Modélisez ce problème comme un CSP et dites :
1. *Quel choix faites-vous pour les variables ?*
2. *Quels sont les domaines des variables ?*
3. *Quelles sont les contraintes qui portent sur ces variables ?*

Énoncé 31 *Considérons le graphe suivant dans lequel on donne :*
- *pour chaque sommet son nom et son domaine,*
- *sur chaque arête, le numéro de la contrainte que cet arc représente sachant que cette contrainte signifie que les deux sommets adjacents ne peuvent pas avoir la même valeur.*



1. *Appliquez le backtrack en prenant les variables par ordre alphabétique et les valeurs par ordre croissant. Vous donnerez uniquement l'arbre d'affectation des variables en précisant les impasses (avec le numéro des contraintes violées) et en arrêtant à la première affectation valide.*
2. *Faire la même chose avec un ordonnancement préalable : ordre croissant sur la taille des domaines, puis si égalité ordre décroissant sur le nb de contraintes et enfin ordre alphabétique.*

3. *Pour chaque variable donnez la valeur la moins contraignante en expliquant.*
4. *Supposons qu'on assigne la valeur 2 à B. Que donnerait une propagation de contraintes jusqu'à ce qu'il n'y ait plus de changement ?*

Énoncé 32 *[Lewis Caroll] 5 personnes de 5 métiers différents dans 5 maisons de couleurs différentes, avec un animal domestique et une boisson préférée distincts.*
On sait que :
- *L'anglais habite dans la maison rouge.*
- *L'espagnol a un chien.*
- *Le japonais est peintre.*
- *L'italien boit du thé.*
- *Le norvégien habite la première maison.*
- *L'habitant de la maison verte boit du café.*
- *La maison verte est après la blanche.*
- *Le sculpteur élève des escargots.*
- *Le diplomate habite la maison jaune.*
- *On boit du lait dans la 3e maison.*
- *La maison du norvégien est à côté de la bleue.*
- *Le violoniste boit du jus de fruit.*
- *Le renard est dans la maison après celle du docteur.*
- *Le cheval est la maison après celle du diplomate.*
- *Le zèbre est dans la maison blanche.*
- *Une des personnes boit de l'eau.*

Les questions auxquelles on doit répondre : Qui habite où ? Qui boit quoi ? Quel est le métier de qui ? Qui a tel animal de compagnie ? Quelle est la couleur de telle maison ?
Modélisez ce pb sous la forme d'un CSP et résolvez-le.

Bilan :
- *CSP* : “Constraint Satisfaction Problem”
- Problème NP-complet
- Méthode de représentation d'un pb à base de *variables* prenant leurs valeurs dans des *domaines* et devant respecter des *contraintes*
- But : attribuer une valeur à chaque variable en respectant les domaines et les contraintes
- Algorithme de base : le *backtrack*
- Amélioration de l'algorithme : ordonnancement des variables (parmi bq d'autres améliorations possibles)

Chapitre 10

Programmation linéaire par nombres entiers (PLNE)

Dans d'autres cours de la formation, ont été abordés divers problèmes de *programmation linéaire* avec des contraintes et des variables en nombres réels (voir par exemple dans le cours d'optimisation).
Une manière d'étendre ce problème est d'exiger qu'un ou l'autre de ses aspects s'exprime en nombres entiers.
Dans le cas où les contraintes et les variables sont toutes deux entières, on parle de programmation linéaire en nombres entiers, PLNE ou simplement programmation en nombres entiers (IP en anglais).
Dans le cas où seulement un de ces aspects s'exprime en nombres entiers, ou même seulement certaines des variables, on parle de programmation linéaire mixte (MP en anglais).
Ici, nous n'aborderons que le cas de la PLNE-IP.
L'exemple suivant illustre le but de cette approche.

Exemple 14 *On veut maximiser la valeur de* $z = x_1 + x_2$ *sachant qu'on doit respecter les contraintes suivantes :*
- $(-2x_1 + 2x_2) \geq 1$
- $(-8x_1 + 10x_2) \leq 13$
- *et* x_1 , $x_2 \geq 0$

En programmation linéaire classique, on trouve l'optimum $\{x_1 = 4, x_2 = 9/2\}$.
Si le problème est contraint en nombres entiers (les x_i *sont des unités indivisibles), alors l'optimum est* $\{x_1 = 1, x_2 = 2\}$.

Cette approche sert pour toute une gamme de problèmes parmi lesquels on retrouve des problèmes d'optimisation déjà évoqués dans ce cours (items 3 et 5 ci-après) :
1. Problèmes avec entrées/sorties discrètes : production d'objets, etc.
2. Problèmes avec conditions logiques : ajout de variables entières avec des contraintes supplémentaires. (par exemple : si le produit A est fabriqué alors produire également B ou C)...
3. Problèmes combinatoires (séquençage, allocation de ressources, emplois du temps, TSP) : formulables en IP.
4. Problèmes non linéaires : souvent formulables en IP. C'est utile en particulier quand la région réalisable est non-convexe.
5. Problèmes de réseaux, problèmes de graphes – exemple : colorier une carte.

Un programme linéaire en nombres entiers correspond donc à un système d'équations et inéquations linéaires (contraintes) dont les inconnues sont à valeurs entières positives ou nulles et les coefficients sont entiers, avec une fonction à optimiser (minimiser ou maximiser), qui est linéaire à coefficients réels. Un programme linéaire est dit sous forme normale, si les contraintes ne sont que des équations. Notons que l'on peut toujours se ramener à une forme normale en ajoutant une inconnue supplémentaire par inéquation, afin de la représenter par une équation.

10.1 Définition formelle

Cette définition est similaire à celle utilisée en Programmation linéaire, mais avec des variables entières :

Définition 16 (Forme canonique d'un pb de PLNE) *Soit un vecteur de variables* $x = (x_1, \ldots, x_n)$, *soit deux vecteurs constants c de* $\mathbb{R}^n$, *b de* $\mathbb{R}^k$, *soit une matrice constante A de* $\mathbb{R}^k \times \mathbb{R}^n$. *Un problème de* programmation linéaire en nombres entiers (PLNE) *consiste à maximiser (ou minimiser) la fonction* $c^T \times x$ *sous la contrainte :* $A \times x \leq b$ *et avec :* $x \geq 0$, $x \in \mathbb{Z}^n$.

Rappel : En mathématiques, la matrice transposée (ou la transposée) d'une matrice $A \in M_{m,n}(K)$ est la matrice $A^T \in M_{n,m}(K)$ obtenue en échangeant les lignes et les colonnes de A.
Par exemple, si $A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$ alors $A^T = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$

Exemple 14 page précédente (cont'd) *Ici on a* $k = n = 2$ *et :*
- $x = (x_1, x_2)$
- $c = (1, 1)$
- $A = \begin{pmatrix} 2 & -2 \\ -8 & 10 \end{pmatrix}$
- $b = (-1, 13)$

10.2 Algorithme de résolution

Point de vue de la complexité Alors qu'il existe des algorithmes polynomiaux pour résoudre les problèmes de programmation linéaire en variables réelles (ce qui établit que ces problèmes sont polynomiaux), le problème de décision associé au problème de programmation linéaire en variables entières est NP-complet.

Méthode de résolution En terme de résolution, remarquons qu'on pourrait, comme pour les CSP, essayer pour chaque variable chaque valeur possible de domaine, et garder la valeur optimale. Mais il y a énormément de valeurs et de combinaisons à essayer. Ce n'est donc pas efficace du tout.
Le principe utilisé correspond à une recherche arborescente avec élagage (appelée “branch and bound” en anglais). On utilise en général la solution réelle fournie par la programmation linéaire “classique” comme point de départ.
- On divise l'espace de recherche pour chaque variable x_i, autour de l'optimum de la relaxation réelle.
- Supposons par exemple qu'on cherche un maximum ($e*$).
- Soit v_i l'optimal réel correspondant à la variable x_i, on pose e_i = partie entière de v_i.
- On explore soit $x_i \leq e_i$, soit $x_i \geq e_i + 1$. Cela nous donne deux nouveaux problèmes avec une contrainte en plus : c'est la partie “branch”.

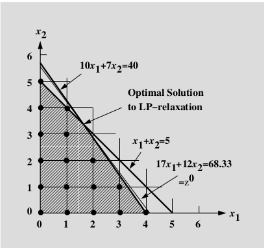
— On relance le calcul de l'optimal réel avec cette nouvelle contrainte sur la branche correspondante.
— Si la solution du programme linéaire du sous-problème e est entière : on a une borne inférieure (faisable) de l'optimum entier ($e \leq e*$). Cette branche est donc un point d'arrêt. C'est la partie "bound".
— Si la solution e fournit une borne supérieure (infaisable) de la suite de l'exploration : on pourra couper les sous-branches non entières qui seront plus grandes. C'est aussi une partie "bound".

Déroulement sur un exemple

Exemple 15 *Considérons le problème suivant :*

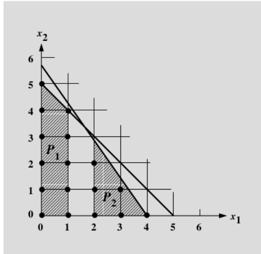
maximiser $17x_1 + 12x_2$
tels que $10x_1 + 7x_2 \leq 40$ *et* $x_1 + x_2 \leq 5$
avec $x_1,\ x_2 \geq 0$ *et entiers*

Si on représente ce problème sous la forme d'un espace cartésien, on a le schéma suivant dans lequel les points représentent les valeurs entières possibles et la zone hachurée l'ensemble des valeurs réelles possibles :



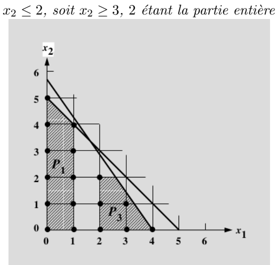
La résolution en programmation linéaire classique produit la solution réelle $(x_1, x_2) = (5/3, 10/3)$, *et la valeur maximisée est égale à* 68.33.

Considérons maintenant x_1 *et appliquons le "branch"; cela nous donne deux nouveaux sous-problèmes* P_1 *(le pb d'origine plus la contrainte* $x_1 \leq 1$, 1 *étant la partie entière de* $5/3$*) et* P_2 *(le pb d'origine plus la contrainte* $x_1 \geq 2$*). Ces deux sous-pbs apparaissent en hachuré sur le schéma suivant :*



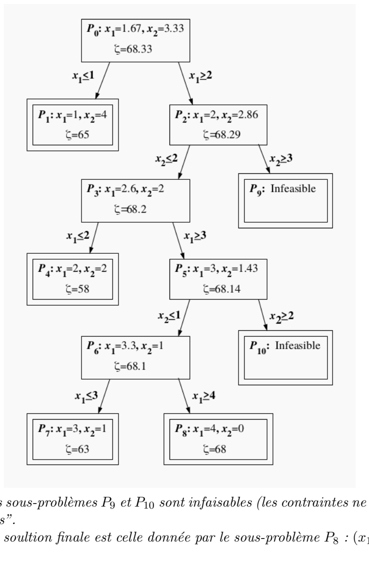
Sur P_1, *la résolution en programmation linéaire fournit une solution qui est entière* $(x_1, x_2) = (1, 4)$ *et qui produit un optimum de* 65. *On a donc obtenu une borne inférieure faisable. Puisqu'on cherche à maximiser, c'est donc inutile d'approfondir cette branche. C'est un "bound".*

Sur P_2, *la résolution en programmation linéaire fournit une solution qui est* $(x_1, x_2) = (2, 20/7)$. *Cette solution n'est pas entière et elle produit un optimum de* $68,29$ *(inférieur à l'optimum obtenu par valeur réelle). On peut donc itérer le processus en raffinant cette zone ("branch") par ajout de contrainte sur* x_2 *(soit* $x_2 \leq 2$, *soit* $x_2 \geq 3$, 2 *étant la partie entière de* $20/7$*).*



Considérons le sous-problème P_3 *construit à partir de* P_2 *et de l'ajout de* $x_2 \leq 2$ *: la résolution en programmation linéaire fournit une solution qui est* $(x_1, x_2) = (2.6, 2)$. *Cette solution n'est pas entière et elle produit un optimum de* $68, 2$. *Nouvelle itération par "branch" avec soit* $x_1 \leq 2$, *soit* $x_1 \geq 3$.

Le schéma suivant donne la représentation de ce processus au complet sous la forme d'une arborescence :



Remarquons que les sous-problèmes P_9 *et* P_{10} *sont infaisables (les contraintes ne sont plus vérifiées) ; ce sont des "bounds".*
Sur cet exemple, la soultion finale est celle donnée par le sous-problème P_8 *:* $(x_1, x_2) = (0, 4)$ *pour un optimum de* 68.

10.3 Exemples d'utilisation

Exemple 16 *[Couverture d'un ensemble] Imaginons qu'on veuille obtenir au moins un exemplaire de tous les morceaux différents enregistrés par un musicien en achetant le moins de disques possibles.*
Par exemple Jimi Hendrix a enregistré environ une soixantaine de chansons, et sa maison de disque a sorti plus de 40 disques à son nom.
Si chaque disque est vu comme un ensemble de morceaux, le problème consiste à trouver un sous-ensemble minimal de disques tel que l'union des morceaux qu'ils contiennent est l'ensemble total des morceaux (en supposant qu'on ne se préoccupe pas d'enregistrements différents de la même chanson).
On a donc ici :
— *Un ensemble de disques* $D = \{d_1, \ldots, d_n\}$.
— *Un ensemble de morceaux* $M = \{m_1, \ldots, m_k\}$.
— *Tels que chaque* $d_i \subset M$. *On suppose aussi que* $\bigcup_i d_i = M$.
— *Et on cherche un sous-ensemble* S *inclus dans* D, *tel que l'union des éléments de* S *(les disques) soit égale à* M.

— *La décision porte ici sur le choix des disques (prendre ou ne pas prendre un disque) :* $x_i = 1$ *on prend le disque* d_i, $x_i = 0$ *on ne le prend pas.*
— *On cherche à minimiser le nombre de disques, donc à minimiser* $\Sigma_i x_i$.
— *Pour tout* m_i, *il faut savoir à quel disque il appartient (donnée du problème). Notons* c_{ij} *la constante disant si le morceau* m_i *est sur le disque* d_j *(*$c_{ij} = 1$ *s'il y est,* $c_{ij} = 0$, *s'il n'y est pas).*
— *La question est donc de savoir combien de disques "choisis" contiendraient le morceau* i, *sachant que ce nombre doit être au moins 1 (puisqu'on veut que chaque morceau soit présent dans la sélection). On doit donc avoir*

$$\Sigma_j(c_{ij} \times x_j) \geq 1$$

On a donc défini notre problème de couverture d'un ensemble sous la forme d'un pb de PLNE :

Soit $D = \{d_i\}$, $M = \{m_i\}$, $C = \{c_{ij}\}$
minimiser $\Sigma_i x_i$
tels que $\Sigma_j(c_{ij} \times x_j) \geq 1$
avec $x \geq 0$, $x \in \mathbb{Z}^n$

10.4 Exercice

Énoncé 33 *Reprendre le problème de coloration exprimé dans l'exemple 1 page 4 et proposer un encodage sous la forme d'un problème de PLNE.*
On considérera que le graphe traité $G = (X, E)$ *est non-orienté.*

Énoncé 34 *Reprendre le problème du voyageur de commerce exprimé dans l'exemple 9 page 7 et proposer un encodage sous la forme d'un problème de PLNE.*
On considérera que le graphe traité $G = (X, E)$ *est non-orienté, complet et d'ordre* n, *le poids sur l'arête* (x, y) *étant noté* p_{xy}.

10.5 Et en pratique ?

Ecrire un ensemble de contraintes peut être fastidieux pour de grands problèmes. Il va donc falloir utiliser un langage de formalisation qui permettra les interactions avec le solveur chargé des calculs et de la résolution.
Il existe plusieurs langages possibles parmi lequel le langage ZIMPL qui est un langage de spécification de problèmes de PNLE capable de fournir à un solveur des descriptions concises et proches du modèle mathématique du problème. Il est notamment compris par le solveur SCIP qui sera utilisé en TP.
Voir [ABKW08, Ach09] pour plus d'information sur SCIP et [Koc04] sur le langage ZIMPL.

Bilan :
• *PLNE* : "Programmation Linéaire en Nombres Entiers"
• Méthode de représentation d'un pb à base d'*équations* et *inéquations mathématiques* portant sur des vecteurs de *variables entières*
• But : trouver des vecteurs de valeurs entières à affecter aux vecteurs de variables satisfaisant les équations et inéquations mathématiques
• Algorithme de base : le *branch and bound*

Bibliographie

[ABKW08] Achterberg (Tobias), Berthold (Timo), Koch (Thorsten) et Wolter (Kati). – Constraint integer programming : A new approach to integrate Cp and MIP. *In : Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, éd. par Perron (Laurent) et Trick (Michael A.). pp. 6–20. – Berlin, Heidelberg, 2008.

[Ach09] Achterberg (Tobias). – SCIP : solving constraint integer programs. *Mathematical Programming Computation*, vol. 1, n° 1, Jul 2009, pp. 1–41.

[AS93] Alliot (Jean Marc) et Schiex (Thomas). – *Intelligence Artificielle et Information Théorique*. – Toulouse France, Cépaduès Éditions, 1993.

[Gal86] Gallier (Jean H.). – *Logic for computer science. Foundations of automatic theorem proving*. – New York, Harper and Row, 1986.

[GJ79] Garey (Michael R.) et Johnson (David S.). – *Computers and Intractability : A Guide to the Theory of NP-completeness*. – New York, W.H. Freeman and Company, 1979.

[Koc04] Koch (Thorsten). – *Rapid Mathematical Programming*. – Thèse de PhD, Technische Universität Berlin, 2004. ZIB-Report 04-58.

[PS82] Papadimitriou (Christos H.) et Steiglitz (Kenneth). – *Combinatorial Optimization : Algorithms and Complexity*. – Prentice-Hall, 1982.

[Ram88] Ramsay (Allan). – *Formal methods in artificial Intelligence*. – Cambridge University Press, 1988.

[Van98] Vanderbei (Robert J.). – *Linear programming - foundations and extensions*. – Kluwer, 1998, *Kluwer international series in operations research and management service*, volume 4.

[Xuo92] Xuong (N.H.). – *Mathématiques discrètes et informatique*. – Masson, 1992.

Index