FILIERE INGENIEUR EN COGNITIQUE

Support de cours de l'UE de 2ème année :

INTELLIGENGE ARTIFICIELLE

PARTIE I

Responsable : Jean-Marc Salotti

Table des matières

1. Introduction à l'intelligence artificielle

1.1. Définition

Il n'existe pas de définition consensuelle du terme « intelligence ». De façon simple et pragmatique, peut être qualifié d'intelligente toute entité capable de résoudre un problème complexe, en se restreignant éventuellement au domaine d'application du problème. Et si l'entité a été fabriquée de manière artificielle. Par exemple, un ordinateur doté d'un logiciel permettant de jouer aux échecs est intelligent pour jouer à ce jeu.
Remarque : un traitement de l'information part d'un ensemble de valeurs appelées les entrées et fournit un autre ensemble de valeurs appelées les sorties. En général, on peut définir un critère permettant d'évaluer « l'intelligence » du traitement. En conséquence, le traitement est intelligent, ou pas, mais on ne peut pas parler de simulation de l'intelligence, car la simulation est également un traitement de l'information. Par exemple, on ne peut pas simuler de bien jouer aux échecs, on joue bien, ou mal, mais il n'est pas possible de simuler un jeu intelligent sans intégrer un traitement intelligent de l'information.

1.2. Histoire de l'I.A.

Mots-clés :
- Problèmes de Hilbert
- Calculabilité
- Thèse de Church Turing
- Machine de Turing
- Test de Turing.
- Marvin Minsky
- Langage LISP, Mc Carthy
- Approche symbolique, systèmes experts.
- Prolog
- Ordinateur 5ème génération au Japon
- Deep Blue bat Kasparov aux échecs
- Théorie des graphes, Dijkstra, A*, Min-Max, etc.
- Réseaux de neurones, connexionnisme
- Robotique

Voir la section 1.3 « The History of Artificial Intelligence » du livre Artificial Intelligence, A Modern Approach de Russel et Norvig [1].

1.3. I.A. faible et I.A. forte

I.A. faible : on peut concevoir des programmes partiellement intelligents, mais il n'est pas possible de concevoir des programmes vraiment intelligents, on ne peut pas obtenir la conscience et la pensée de façon algorithmique.
I.A. forte : on peut concevoir des programmes qui auront autant de conscience et de personnalité que des humains. Principes fondateurs de l'I.A. forte :
- Le fonctionnement du cerveau, c'est-à-dire l'état interne et les signaux d'entrées sorties, peut être décrit par un algorithme prenant en compte ses éléments, son organisation et les règles physico-chimiques.
- L'algorithmique intègre tous les mécanismes de calculabilité mathématique => Tout ce qui est calculable est calculable par un ordinateur (thèse de Church-Turing).

Chambre chinoise :
La chambre chinoise est une expérience de pensée de John Searle, philosophe, proposée en 1980 : une personne ne comprenant pas le Chinois est isolée dans une chambre. Cette personne exploite un catalogue de règles (un algorithme), qui serait la synthèse des connaissances et des règles d'exploitation de ces connaissances d'un vrai Chinois, pour répondre à une question écrite en Chinois sur une feuille de papier. Selon l'I.A. forte, cette personne serait en mesure de répondre à n'importe quelle question. La personne ne comprenant pas le Chinois et le catalogue ne pouvant prétendre à une conscience, cela invaliderait la thèse de l'I.A. forte.
La conclusion de Searle a été abondamment critiquée. Principales critiques :

- Il faudrait des milliards de règles pour y parvenir, y compris des règles permettant l'apprentissage de nouvelles règles.
- Le fonctionnement du cerveau peut également être décrit par un catalogue de règles, donc le système composé de la personne qui manipule le catalogue et le catalogue lui-même aurait probablement une conscience.

Exemples de problèmes difficiles en I.A. :

a. **Capacité de projection et de raisonnement complexe en mobilisant des connaissances générales non indiquées dans l'énoncé d'une question.**
   Exemple : Demain, j'ai rendez-vous pour déjeuner avec Napoléon Bonaparte, vrai ou faux et pourquoi ?
b. **Capacité d'apprentissage complexe :**
   Exemple : soient 3 objets appelés « aba », « baa » et « cab ». Un objet est dit « zorglub » d'un autre objet si son nom est une anagramme du nom de cet autre objet. Un baa est-il le zorglub d'un aba ?
   Remarque : Nouvelles données et nouvelle règle à acquérir => règles sur catalogue figé insuffisante.
c. **Connaissances difficiles à expliciter :**
   Exemple : « Donnez-moi une description du visage de Einstein pour pouvoir l'identifier dans une base de données d'images. » Certaines connaissances sont codées dans le cerveau de manière non consciente et sont très difficiles à expliciter => I.A. symbolique insuffisante

b et c suggèrent que l'approche connexionniste est nécessaire pour concevoir une I.A. forte.

Position intermédiaire entre I.A. faible et I.A. forte : il existe théoriquement des algorithmes capables de générer l'intelligence alliée à une intention et une conscience mais il est peut-être impossible de les concevoir en pratique tant le problème est complexe.

1.4. Domaines de l'I.A. ou en lien étroit avec l'I.A.

- Vision par ordinateur
- Reconnaissance des formes
- Perception
- Reconnaissance de la parole
- Raisonnement automatique
- Logique, logique floue, logique temporelle, logique du premier ordre
- Systèmes experts
- Agents, systèmes multi-agents
- Théorie de la décision
- Arbres de décision
- Fouilles de données, data mining
- Traitement automatique du langage
- Attention, motivations, émotions
- Apprentissage symbolique
- Apprentissage par réseaux de neurones
- Classification automatique
- Connexionnisme
- Résolution générale de problèmes
- Planification
- Théorie des graphes
- Théorie des jeux
- Robotique
- Architectures de contrôle en robotique
- Robotique développementale
- Soft Computing
- Algorithmes génétiques

2. Introduction à la robotique

2.1. Principales liaisons

- Liaison pivot

1 seul degré de liberté (DOF=Degree Of Freedom) : 1 rotation
Exemple : articulation du genou, pédale de vélo, roue et axe
C'est la plus utilisée en robotique. Un servomoteur avec 1 axe de rotation est classiquement utilisé.

- Liaison glissière

1 seul degré de liberté : 1 translation
Classiquement utilisé pour les grues, ainsi que dans les hangars avec des ponts suspendus.

2.2. DOF pour un bras robotique

Le nombre de degrés de liberté est généralement égal au nombre de liaisons pivot.
Problème classique : combien de liaisons pivot faut-il pour que la pince d'un bras robotique puisse toucher n'importe quel point placé à côté du robot ?
Réponse : 3
Problème complémentaire : combien de liaisons pivot supplémentaires faut-il pour positionner la pince selon n'importe quelle orientation ?
Réponse : 3
Nota Bene : il faut 6 degrés de liberté pour qu'un bras manipulateur puisse accéder à tous les points placés devant lui et puisse positionner la pince selon n'importe quelle orientation.
Et un autre degré de liberté pour ouvrir/fermer la pince !

Exemples de robots manipulateurs :

- 4 DOF : robot manipulateur le plus simple. Le robot peut placer la pince n'importe où, mais il est obligé de prendre l'objet verticalement.
- 5 DOF, exemple du robot Lynxmotion ci-dessous : on ne peut pas prendre un objet en mettant le poignet en travers. 5 servomoteurs pour la base, l'épaule, le coude, le poignet (2 rotations) et un 6ème servomoteur pour la fermeture.

- Robot NAO, dont 2 bras et 2 jambes : 25 DOF

Remarque : Biomécanique. Un être humain dispose de plusieurs centaines de muscles et de 206 os, ce qui offre plus de cent degrés de liberté. De plus, les liaisons comportent souvent un peu de souplesse, ce qui augmente encore les degrés de liberté et donc les positions possibles.
Question : où se situent les principales différences entre un humain et NAO pour ce qui concerne les DOF ?
Réponse : les mains et la colonne vertébrale.

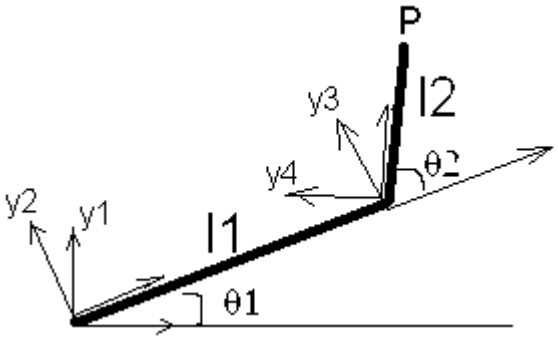
2.3. Cinématique

Problème classique : connaissant les angles pour chaque articulation, quelle est la position x,y,z de la pince dans le repère du sol ?
Rappels mathématiques :
Soit un servomoteur permettant la rotation d'une partie du robot autour d'un axe z. On attache un repère à la partie qui ne bouge pas et un autre à la partie qui est en rotation, mettons d'un angle a. Le passage d'un repère à un autre est donné par la matrice suivante :

(cos a - sin a 0)
(sin a cos a 0)
(0 0 1)

$P_{R2} = M\ P_{R1}$

Exercice :



Déterminer la position du point P(x,y) après les 2 rotations, en fonction des angles et de l1 et l2.

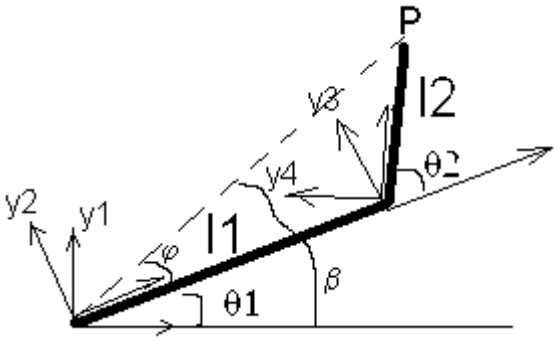
2.4. Cinématique inverse

Le problème de cinématique inverse est le problème fondamental du roboticien qui cherche à déterminer les angles qu'il doit attribuer à chaque articulation pour atteindre la position désirée.
Considérations générales : soit une position P(x,y,z) à atteindre par la pince dans le repère du robot. Quelles valeurs donner aux angles ?
En général, placer le bras du robot en face du point à atteindre est aisé car il suffit de faire tourner le robot autour de sa base et il n'y a qu'un seul angle à trouver. Le problème se ramène alors à un plan et il ne reste plus que 2 angles à trouver.

- Résolution algébrique :

Equations difficiles !
Astuce : pour un angle a donné, on exprime cos(a) et sin(a) en fonction de tan(a/2)
u = tan (a/2)
=> $\cos(a) = (1-u^2)/(1+u^2)$
=> sin(a) = 2u/(1+u2)

- Résolution géométrique



$x^2+y^2 = l1^2 + l2^2 - 2l1\ l2 \cos(180-\theta2)$
⇨ θ2
Pour θ1 :
θ1 = β +- φ (2 solutions)
β = atan2(y,x) (atan2 est l'arc tangente à 2 arguments)

puis $\cos(\varphi) = \dfrac{(x^2 + y^2 + l1^2 - l2^2)}{2\ l2\sqrt{x^2 + y^2}}$

S'il faut ensuite imposer un angle de prise ϕ (robot 4 DOF) et déterminer l'angle θ3 du poignet
=> $\theta1 + \theta2 + \theta3 = \phi$ => on déduit ϕ

Remarques :
Il existe souvent plusieurs solutions possibles pour placer la pince au bon endroit.
Si le nombre de ddl du robot est inférieur au nombre de ddl de l'espace opérationnel, il peut ne pas y avoir de solution.
S'il n'y en a que 2, le choix est relativement simple. Il peut obéir à une loi simple (θ1 max par exemple)
S'il y en a une infinité (trop de degrés de liberté), le problème peut s'avérer ardu. Problèmes classiques : obstacle potentiel, passage par un mouvement interdit, …
Critères possibles pour sélectionner la solution :

- Réduction arbitraire des degrés de liberté
- Minimisation des déplacements du bras
- Minimisation de l'énergie requise pour actionner le bras

Exercice : Donnez les spécifications techniques d'un bras manipulateur comportant uniquement des liaisons pivot et dont la seule fonction est de prendre des crayons posés à plat sur une table, sachant qu'il n'y a pas d'autres objets. Quel est le nombre minimal de servomoteurs ? Justifiez votre réponse.

2.5. Capteurs

Liste de capteurs :

- Odomètre : compteur de tours de roues
- Contact, pression, texture …
- Ultrasons (permet une estimation grossière de la distance)
- IR, passif ou actif
- Ondes radio (blue tooth, WIFI, …)
- Laser (calcul de distance en fonction du délai du retour)
- Caméra couleur, N&B, IR, multispectrale, rayons X, …
- Microphone
- GPS (ondes radio + satellites + horloge + calculs)
- Puce RFID (radio frequency identification)
- Code-barre
- Boussole
- Accéléromètre (capteur de force)
- Gyromètre (vitesse angulaire)
- Inclinomètre
- Chronomètre / horloge

2.6. Incertitudes en robotique

- Détection

Incertitudes sur la mesure des capteurs (problème de précision)
Incertitudes sur ce qu'a détecté ou pas le capteur (cas particulier de l'environnement)

- Action

Incertitudes sur l'action effectuée par un servomoteur (pb de précision sur la rotation demandée, vitesse et accélération)
Incertitudes sur les dimensions du robot (précision de la mesure sur le diamètre de la roue)
Incertitudes sur la relation entre le robot et l'environnement (glissements, frottements)

- Représentation, décision

Incertitudes sur la position du robot
Incertitudes sur l'état de l'environnement (porte ouverte ou fermée, position de la chaise …)
Informations incomplètes (présence d'un objet inconnu)
=> Le roboticien est un expert en gestion des incertitudes !

2.7. Challenges et perspectives en robotique

Il y a eu quelques avancées majeures ces dernières années, avec par exemple les techniques de « SLAM » (Self Localization and Mapping) qui permettent à un robot de découvrir un environnement, de le modéliser et de s'y repérer en temps réel. Cela reste toutefois une tâche très difficile. La reconnaissance visuelle et la reconnaissance de la parole des robots sont également un cran en dessous des performances humaines. Au niveau attentionnel et interactions sociales, le challenge reste entier. Enfin, il ne faut pas négliger les problèmes d'autonomie énergétiques.

3. Algorithmes génétiques

Les algorithmes génétiques sont classiquement utilisés pour résoudre des problèmes d'optimisation, typiquement caractérisés par la recherche du meilleur ensemble de valeurs à attribuer à plusieurs variables constituant un vecteur.

Exemples :

- Trouver les positions des reines dans le problème classique des 8 reines à placer sur un échiquier sans qu'aucune ne soit en prise avec une autre.
- Trouver les vitesses initiales en norme et en orientation d'un ensemble de projectiles pour maximiser les dégâts occasionnés.
- Trouver les conditions initiales et les règles de survie du jeu de la vie qui convergent vers une forme particulière.

Fondamentalement, la recherche est locale. La solution est en effet trouvée par des déplacements élémentaires dans l'espace d'état. Il n'y a aucune garantie qu'une solution optimale soit trouvée.

Principes généraux des algorithmes génétiques :

Il faut dans un premier temps proposer une modélisation du problème. Il s'agit classiquement de choisir un ensemble de variables clés et les valeurs qui peuvent leur être attribuées.

Comme pour l'évolution des espèces selon la théorie de Darwin, on part d'une population importante d'individus (ici de vecteurs) tous différents. Les vecteurs vont jouer le rôle du code génétique. L'objectif est de faire évoluer cette population pour converger vers des vecteurs optimisés relativement à un critère donné. Le choix de ce critère est très important et conditionne l'évolution de la population.). L'algorithme fonctionne de manière itérative selon le processus suivant :

- Evaluation de chaque individu selon le critère choisi
- Sélection des meilleurs vecteurs (on supprime une partie de la population).
- On génère de nouveaux vecteurs en procédant à des « mutations » sur les vecteurs restants. Par exemple, on choisit une des variables du vecteur et on modifie sa valeur aléatoirement. En général, on reconstitue une population de taille égale à la taille d'origine en gardant les meilleurs de l'étape précédente et en générant de nouveaux vecteurs par mutation. Plutôt que de procéder à des mutations aléatoires, on peut aussi procéder à des croisements de 2 vecteurs.

et ainsi de suite jusqu'à que les vecteurs finaux soient suffisamment « adaptés ».

Remarque 1 : la technique dite du "**recuit simulé**" ou simulated annealing en Anglais est très proche des algorithmes génétiques. Voir section 6.4.

Remarque 2 : la « **programmation génétique** » consiste à modifier des éléments algorithmiques plutôt que des vecteurs ; on teste alors l'efficacité des algorithmes obtenus par mutation de ces éléments.

Exercice :

Le problème des 8 reines consiste à placer 8 reines sur un échiquier de 8x8 cases sans qu'aucune ne soit en prise avec une autre (pas la même ligne, pas la même colonne et pas la même diagonale). Pour résoudre le problème des 8 reines, peut-on exploiter un algorithme génétique ? Si oui, préciser la mise en œuvre et sinon pourquoi.

4. Structures de données de type graphe

4.1. Concepts

Les graphes sont utilisés pour représenter des relations entre les données, sans qu'il y ait nécessairement de hiérarchie comme c'est le cas pour les arbres. On utilise par exemple un graphe pour représenter un réseau routier entre les villes. Les villes sont les données et la relation est l'existence d'une route qui joint 2 villes. Un arbre est en fait un graphe particulier dans lequel il existe systématiquement une relation de type parental, avec un nœud père et des nœuds fils. Il existe également des **graphes conceptuels** permettant d'exprimer des relations complexes entre des personnes, des objets et des actions.
On définit les concepts suivants :

- Nœud ou sommet du graphe : il correspond à la notion de nœud pour les arbres. Chaque sommet est typiquement une instance d'une classe en langage objet.
- Arête du graphe : relation définie par un couple de sommets du graphe. Il peut y avoir plusieurs types de relation dans un même graphe. On associe également parfois une valeur à la relation, par exemple une distance pour un graphe représentant un réseau routier.
- Arc : si la relation n'est pas symétrique (a R b différent de b R a), on parle de **graphe orienté** et les relations sont des arcs. Les 2 sommets de la relation forment l'extrémité initiale et l'extrémité terminale de l'arc.
- Graphe : Pour définir un graphe, on donne généralement l'ensemble de ses sommets et l'ensemble de ses arêtes ou arcs. Ex. : G = { {S1;S2;S3;S4} ; {(S1,S2);(S1,S3);(S2,S4)}}
- Sous-graphe : Un sous-graphe est un sous-ensemble de sommets et un sous-ensemble d'arêtes d'un graphe, chacune d'elle ne faisant référence qu'à un des sommets du sous-ensemble.
- Cycle : Un cycle existe dans un graphe orienté s'il existe une suite d'arcs permettant de partir d'un sommet et de revenir sur ce même sommet.
- Graphe connexe : Un graphe est dit connexe s'il est possible de trouver un chemin (une suite d'arêtes) permettant de relier n'importe quel couple de sommets.
- Isomorphisme de graphe : 2 graphes sont isomorphes s'il est possible de mettre en correspondance bijective les sommets du 1er avec les sommets du 2ème et d'avoir les mêmes relations entre les sommets.

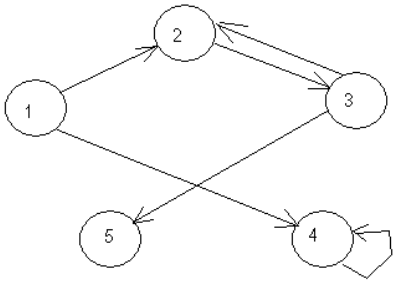
4.2. Structures de données

Il existe un grand nombre de graphes différents, il est par conséquent difficile de proposer une structure générique. Néanmoins, si on a une liste de N sommets, il suffit de les numéroter de 1 à N et de représenter le graphe sous la forme d'une matrice NxN (en fait un tableau de dimension 2) qui mémorise la présence ou l'absence d'une relation ou éventuellement une valeur attachée. C'est la "matrice d'adjacence". S'il faut mémoriser certaines informations pour chaque sommet, on peut définir une structure pour les sommets et retenir le numéro.

Exemple, soit le graphe suivant : { S={1; 2; 3; 4; 5} A={(1,2);(1,4);(2,3);(3,2);(3,5);(4,4)} }
Matrice d'adjacence (0 pas d'arête, 1 arête) :

	1	**2**	**3**	**4**	**5**
1	0	1	0	1	0
2	0	0	1	0	0
3	0	1	0	0	1
4	0	0	0	1	0
5	0	0	0	0	0

Dessin du graphe :



La création d'un graphe tel qu'il a été défini ne pose pas de problème fondamental, il faut donc simplement initialiser une liste et remplir toutes les cases de la matrice d'adjacence.

Si on ne connaît pas la taille du graphe (on est en train de l'explorer) ou si le graphe est très grand, la matrice d'adjacence n'est sans doute pas la structure de données la plus adaptée. Dans ce cas, on peut exploiter les listes. Par exemple, la liste principale contient les nœuds et chaque nœud contient à son tour une liste de relations, une relation étant définie par un nœud initial, un nœud arrivée et éventuellement une information associée. Cette structure de donnée est toutefois plus complexe à mettre en œuvre.

Exercices :

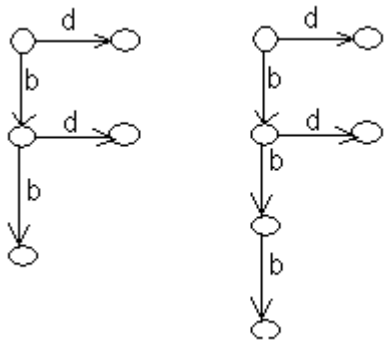
1) Compter le nombre de sommets initiaux

Soit un graphe orienté défini par une matrice d'adjacence déclarée comme tableau à 2 dimensions de booléens (dans un langage quelconque). Le premier indice indique le nœud de départ et le deuxième le nœud d'arrivée. Il y a true dans chaque case du tableau correspondant à un arc. Ecrire une fonction qui prend en paramètre le tableau ainsi que sa taille et compte le nombre de sommets qui sont le départ d'un arc. Dans l'exemple précédent comportant 5 nœuds, cette fonction doit retourner la valeur 4.

2) Reconnaissance des formes

En général, la reconnaissance de formes avec des graphes nécessite la recherche d'isomorphisme de graphes ou sous-graphes, ce qui est très complexe. On s'intéresse ici à une méthode plus simple.

On représente des lettres de l'alphabet par des graphes de la façon suivante. Les sommets du graphe correspondent aux jonctions et aux extrémités des traits qui forment les lettres. Les relations du graphe sont "d" pour à droite ou "b" pour vers le bas. 2 exemples pour le F :

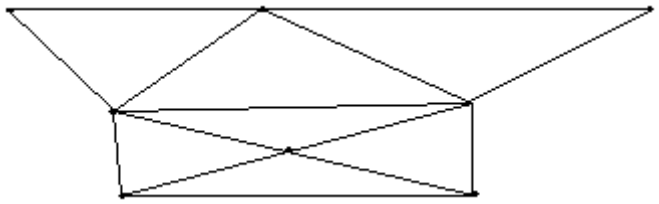


Pour mémoriser les relations, la matrice d'adjacence contient ici des caractères.

Supposons que la lettre soit **F**, **E**, **H** ou **T**. Ecrire une ou plusieurs fonctions/procédures permettant de distinguer chaque cas.

3) Propriété des graphes

Est-il possible de dessiner le graphe ci-dessous avec un crayon sans lever la main et sans retracer la même relation ? Trouvez un moyen simple pour décider de la faisabilité.



5. Résolution générale de problèmes

5.1. Conceptualisation

Exemples de problèmes :

- Quel est le plus court chemin pour aller de Bordeaux à Strasbourg en voiture ?
- Aux échecs, étant donnée une situation de jeu, quel est le meilleur coup possible ?
- Comment construire un radeau avec des bouts de bois ?
- Pour un déménagement, quel est le rangement dans le camion qui minimise la place occupée ?
- Au jeu du Rubik's cube, quels mouvements faut-il faire pour reformer 6 faces de couleur homogène ?

Comme les 4 problèmes précédents, de nombreux autres problèmes peuvent être définis grâce aux 4 points suivants :

- **Etat Initial** :

L'état initial est constitué d'un environnement de départ et d'hypothèses initiales.

- **Liste d'actions possibles** :

Une liste d'actions est proposée sous la forme d'opérateurs ou de fonctions permettant d'évoluer à partir d'un état donné vers un autre état.
En règle générale, on dispose donc d'une "**fonction successeur**", qui a pour argument un état X et qui retourne une liste de couples (action, état successeur) pour chaque action possible à partir de X. L'état initial et la fonction successeur définissent l'**espace d'état du problème**. Cet espace d'état forme un graphe dans lequel les noeuds sont les états et les arcs entre les noeuds sont les actions. Un **chemin** dans cet espace d'état est une séquence d'états obtenus après une séquence d'actions.

- **Etat final** :

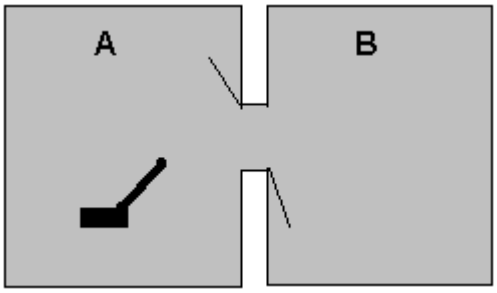
L'état final, solution au problème posé, est identifié par une fonction booléenne appelée **test d'état final**. Il peut y avoir plusieurs états finaux. Le test peut également être caractérisé par une propriété abstraite. Par exemple, aux échecs, l'objectif de "pat" (égalité) n'est pas formellement caractérisé, il reste relativement subjectif.

- **Coût d'un chemin** :

A chaque chemin, complet ou partiel, on peut attribuer un indice de performance. On exploite généralement une fonction d'estimation du coût d'un chemin qui est inversement proportionnel à la performance. Ce coût est généralement une simple valeur numérique. Pour faciliter la résolution du problème, on choisit une fonction de coût toujours positive égale à la somme des coûts de chaque action le long du chemin considéré.

Une solution au problème est donc un chemin entre un état initial et un état final. La qualité de la solution est mesurée par le coût du chemin. La solution optimale correspond au chemin de moindre coût parmi tous les chemins possibles.

5.2. Exemple avec le travail d'un aspirateur



Un appartement de 2 pièces A et B doit être nettoyé. Initialement, A et B sont sales et l'aspirateur est en A. Les actions possibles sont : "nettoyer la pièce" et "passer dans l'autre pièce". Quelle est la suite d'actions à effectuer pour se retrouver avec 2 pièces propres ?

Modélisation :

Etat initial : SSA (A sale, B sale et aspirateur en A)
Actions possibles : nettoyer ou changer de pièce;
pour nettoyer : si la pièce était sale, elle devient propre, sinon elle reste propre
pour changer de pièce : si on était en A, on passe en B et si on était en B on passe en A
Etat final : PPA ou PPB
Coût d'une action : 1 (on peut éventuellement différencier)

5.3. Recherche de solutions

La recherche de solutions est définie par un algorithme général, donné ci-dessous :

FONCTION Arbre_de_Recherche (problème, *strategie*) : retourne la solution, ou l'échec de la recherche
    Initialisation de la recherche avec l'état initial
    Boucle :
        SI il n'y a plus de feuille candidate pour le développement de l'arbre de recherche, alors échec
        SINON choisir une feuille pour l'expansion de l'arbre en fonction de la *stratégie*
            SI celle-ci correspond à un état final, retourner la solution trouvée
            SINON développer les noeuds successeurs et les ajouter à l'arbre de recherche

- Un aspect fondamental de l'algorithme est la stratégie de développement de l'arbre de recherche. La liste des feuilles de l'arbre correspond à tous les états qui n'ont pas encore été étudiés. Lorsqu'une feuille est "développée", celle-ci est retirée de la liste et ses successeurs y sont ajoutés. La "stratégie" est la méthode qui va déterminer quelle est la feuille qui va être développée en premier parmi la liste des feuilles à développer. Nous appellerons cette liste O pour "Ouverts" car elle représente les parties ouvertes du problème.

5.4. Mesure de performance

La performance de l'algorithme général de résolution de problème est évaluée selon plusieurs paramètres :

- Complétude : l'algorithme garantit-il la découverte d'une solution lorsqu'elle existe ?
- Optimalité : l'algorithme garantit-il la découverte de la solution optimale, c'est à dire celle qui correspond au chemin de moindre coût ?
- Complexité en temps : en combien de temps l'algorithme trouve t-il la solution ?
- Complexité spatiale : quel espace mémoire est requis pour réaliser la recherche ?

Les 2 dernières mesures varient en fonction des données du problème. Un paramètre fondamental est la taille du graphe représentant l'espace d'état. Or, celui dépend essentiellement de 3 paramètres :

1) b, le nombre maximum de branchements, c'est à dire le nombre de successeurs d'un noeud donné
2) d, la profondeur maximale d'un chemin solution (d pour depth)
3) m, la longueur maximale qu'un chemin peut atteindre dans l'espace d'état

Nous exprimerons la complexité des algorithmes proposés par la suite en fonction de ces 3 paramètres.

5.5. Différentes stratégies de recherche

Recherche en largeur d'abord :

La stratégie de développement est relativement simple : on développe les noeuds en fonction de leur date d'arrivée dans la liste O. La liste est ainsi gérée en FIFO (First In First Out).

Exemple : au jeu du Rubik's cube, on place dans O les noeuds correspondant aux états du cube obtenus à partir de la liste des rotations élémentaires possibles (6 cas). Puis, pour chacun des nouveaux états, on supprime le nœud correspondant de la liste O et on envisage la nouvelle liste de mouvements possibles qui conduisent à de nouveaux nœuds à insérer en dernière position dans la liste O. On procède ainsi de manière progressive, jusqu'à ce qu'on atteigne une profondeur donnée (une séquence de rotations en nombre limité) ou un état final.

Recherche en profondeur d'abord :

On développe les derniers noeuds arrivés d'abord. La liste est ainsi gérée en LIFO (Last In First Out).

Exemple : au jeu du Rubik's cube, on place dans O les noeuds correspondant aux états du cube obtenus à partir de la liste des rotations élémentaires possibles (6 cas). On prend alors le premier nœud, on le retire de O puis on détermine les noeuds successeurs qu'on insère dans O en première position. Et ainsi de suite. On traite toujours le nœud qui est en tête de la liste O, de sorte qu'on explore une longue séquence de mouvements avant d'envisager un premier mouvement différent, c'est la recherche en profondeur d'abord. Dans le cas du Rubik's cube, on peut remarquer qu'il n'y a pas intérêt à revenir sur un état déjà vu, car sinon, on rentre dans une boucle infinie sans jamais tomber sur la solution. On peut alors décider de restreindre la recherche à une profondeur donnée.

La recherche peut être faite de manière récursive ("back-tracking search"), ce qui permet de limiter l'espace mémoire utilisé.

Recherche avec le meilleur d'abord :

On développe le noeud qui semble le plus intéressant, c'est à dire celui qui a le coût minimal d'abord. La liste est ainsi triée en fonction du coût associé à chaque noeud. Pour éviter un tri répétitif, on peut insérer les successeurs dans O directement au bon endroit dès qu'ils sont définis.

Exemple : au jeu du Rubik's cube, on place dans O les noeuds correspondant aux états du cube obtenus à partir de la liste des rotations élémentaires possibles (6 cas) et on évalue chaque mouvement. Dans ce jeu, chaque mouvement est une rotation élémentaire, il n'y a donc pas de différence particulière entre les successeurs. On peut associer par exemple un coût de 1 à chaque rotation. Le coût d'un chemin est alors directement proportionnel au nombre de rotations élémentaires envisagées. Si on trie les nœuds de O, on est donc amené à

choisir en premier les nœuds correspondant aux chemins les plus courts, autrement dit, l'algorithme est ici équivalent à un parcours en largeur dans l'espace d'état. On peut toutefois envisager un parcours très différent si on associe un coût différent à chaque rotation, par exemple en considérant que le cube ne tourne pas bien selon un des axes et qu'il faut donc éviter ce mouvement autant que possible.

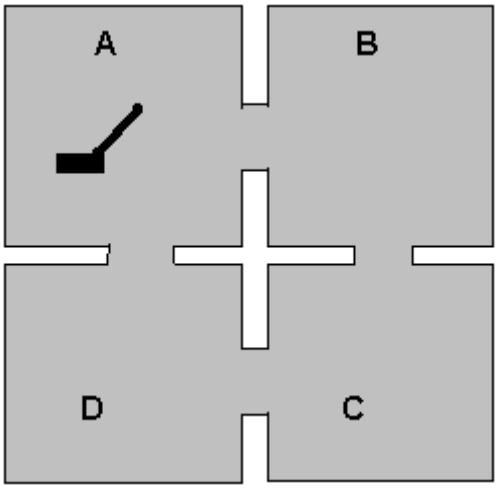
Autres stratégies de recherche :

Il existe d'autres stratégies comme la recherche "en profondeur limitée", la recherche "itérative en profondeur limitée", la recherche "bidirectionnelle", etc. etc.

5.6. Evitement des états revisités

Il arrive fréquemment que des chemins différents conduisent à un même état. Par exemple, aux échecs, certaines séquences d'actions conduisent au même état et sont donc commutatives, idem pour le rubik's cube. Or, l'algorithme proposé ne s'en soucie guère et peut donc développer plusieurs fois un état donné sur des branches différentes de l'arbre de recherche, ce qui est une perte de temps. Pour éviter le développement d'un noeud correspondant à un état déjà étudié, il faut mémoriser la liste des états déjà rencontrés et développés. Nous appellerons F la liste des noeuds associés à ces états, F pour "fermés", c'est à dire déjà vus et déjà traités. Lors du développement d'un noeud, la liste des états successeurs est donc dans un premier temps étudiée et seuls sont insérés dans O et ajoutés à l'arbre les noeuds dont l'état n'est pas représenté par un noeud dans F (ou dans O !). L'arbre devient en réalité un graphe, car on ajoute généralement un arc entre le noeud courant et le noeud de l'arbre situé sur une autre branche qui correspond à un état successeur déjà visité.

5.7. Exercice



Reprenez le problème de l'aspirateur avec l'environnement proposé ci-dessus et développez l'arbre d'exploration de l'espace d'états

a) Avec la technique en profondeur d'abord (discutez l'implémentation de la fonction successeur)

b) Avec la technique en largeur d'abord

c) Avec la technique le meilleur d'abord, en développant en premier l'état qui a le plus de pièces propres.

6. Algorithmes d'exploration avec heuristiques

6.1. Algorithme de Dijkstra : exploration à partir du meilleur d'abord

NB : Historiquement, l'algorithme de Dijkstra permet de trouver les plus courts chemins d'un nœud donné d'un graphe vers tous les autres nœuds du graphe. De plus, l'algorithme de base effectue ce traitement sur un graphe connu à l'avance, ce qui permet de travailler sur des tableaux statiques. Dans la version présentée ici, applicable à la résolution générale de problèmes, l'objectif est de trouver le plus court chemin à partir d'1 seul nœud initial vers un ou plusieurs autres nœuds finaux en explorant dynamiquement les chemins possibles (« Dijkstra dynamique »), c'est-à-dire en définissant les nœuds de l'arbre d'exploration au fur et à mesure. Par analogie, pour trouver le plus court chemin entre Bordeaux et Paris, il n'est pas nécessaire de rentrer la carte routière du monde entier, on peut s'informer des nœuds à explorer (les nouvelles intersections) au fur et à mesure de l'avancement de l'exploration.

Principe de l'algorithme :

On considère que les nœuds correspondent à des états de l'espace d'états du problème et sont définis par une classe (éventuellement abstraite) GenericNode, avec les propriétés suivantes :

```
string Name;
double GCost;          //coût du chemin du noeud initial jusqu'à ce nœud
double HCost ;         // Pas utile pour Dijkstra, mais utile pour A*, voir section 4.2
GenericNode ParentNode;   // noeud parent
List<GenericNode> Enfants;  // noeuds enfants
```

Pour les besoins de l'exploration, on définit 2 listes, la liste O des nœuds "ouverts" (nœuds qui n'ont pas encore été explorés) et la liste F des nœuds "fermés" (déjà explorés). Au début, F est vide et O contient le nœud initial.

On suppose qu'il existe des méthodes dédiées au problème ayant les caractéristiques suivantes :

- GetListSucc (N) : A partir d'un nœud N quelconque, détermine et renvoie la liste des nœuds successeurs (donc les états accessibles), c'est-à-dire ceux qui sont directement reliés à N.
- EndingNode (N) : Détermine si un nœud N est un nœud final (retourne un booléen).
- GetArcCost (N1, N2) : Pour 2 nœuds N1 et N2 reliés par un arc et passés en paramètres, renvoie la valeur associée à l'arc.

Remarque : Lors de l'exploration d'un nœud N, on obtient une liste de ses successeurs. Or, un ou plusieurs de ces nœuds peuvent déjà avoir été rencontrés lors d'une étape précédente, lors de laquelle un autre chemin était exploré. Si c'est le cas, ce nœud serait dans O ou dans F. Pour le savoir, il faut pouvoir reconnaître le nœud en lui associant toute information jugée utile et caractéristique de l'état qu'il représente. Une fonction booléenne est alors utile pour pouvoir comparer 2 nœuds et déterminer s'ils correspondent à un même état.

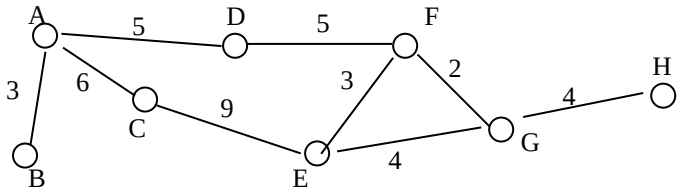
La recherche du plus court chemin se fait selon le principe du meilleur d'abord. On va progressivement faire passer tous les sommets de O dans F. A chaque étape, on choisit dans O le nœud N dont la distance à N0 (nœud initial) est la plus faible et on le passe dans F. On examine alors les nœuds voisins et on construit ainsi petit à petit l'arbre de recherche en insérant les nouveaux nœuds dans les ouverts, ou éventuellement en mettant à jour la distance et le prédécesseur si le nœud avait déjà été rencontré et que le nouveau chemin est meilleur. Lorsqu'on rencontre un nouveau nœud, 2 informations importantes doivent être prises en compte : le coût total du chemin aboutissant à ce nœud depuis le nœud initial et le nœud prédécesseur ayant permis d'y arriver. Cette dernière information permet de retracer tout le chemin une fois qu'une solution est trouvée (nœud final rencontré).

Algorithme

```
// Le noeud passé en paramètre est supposé être le noeud initial
public List<GenericNode> RechercheSolutionAEtoile (GenericNode N0)
{
    L_Ouverts = new List<GenericNode>();
    L_Fermes = new List<GenericNode>();
    GenericNode N = N0;
    L_Ouverts.Add (N0);

    // tant que le nœud n'est pas terminal et que ouverts n'est pas vide
    while (L_Ouverts.Count != 0 && N.EndState() == false)
    {
       // Le meilleur noeud des ouverts est supposé placé en tête de liste
       // On le place dans les fermés
       L_Ouverts.Remove(N);                L_Fermes.Add(N);
```

```
      // Il faut trouver les noeuds successeurs de N
      this.MAJSuccesseurs(N);  // voir plus loin
      // Inutile de retrier car les insertions sont ordonnées

      // On prend le meilleur, donc celui en position 0
      // A condition qu'il existe bien sûr
      if (L_Ouverts.Count > 0){ N = L_Ouverts[0];} else { N = null;}
     }
     // A* terminé. On retourne le chemin sous forme d'une liste de nœuds.
     // Le chemin est retrouvé en partant du nœud final et en accédant
     // aux parents de manière itérative jusqu'au nœud initial.
     List<GenericNode> _LN = new List<GenericNode>();
     if (N != null)
     {   LN.Add(N);
         while (N != N0)
         {  N = N.GetNoeud_Parent();   // Retourne le prédécesseur
            LN.Insert(0, N);  // On insère en première position
         }
     }
     return _LN;
}

private void MAJSuccesseurs(GenericNode N)
{  /* On fait appel à GetListSucc, méthode abstraite qu'on doit réécrire
     pour chaque problème. Elle doit retourner la liste complète des nœuds
     successeurs de N. */
    List<GenericNode> listsucc = N.GetListSucc();
    foreach (GenericNode N2 in listsucc)
    {
       // N2 est-il une copie d'un nœud déjà vu et placé dans les fermés ?
       GenericNode N2bis = ChercheNodeDansFermes(N2.GetNom());
       if (N2bis == null)
       {
          // Rien dans les fermés. Est-il dans les ouverts ?
          N2bis = ChercheNodeDansOuverts(N2.GetNom());
          if (N2bis != null)
          {
             // Il existe, donc N2 n'est qu'une copie de N2Bis
             // Le nouveau chemin N0->N2 passant par N est-il meilleur ?
             if (N.GetGCost() + N.GetArcCost(N2) < N2bis.GetGCost())
             {
             // Mise à jour de N2bis
             N2bis.SetGCost(N.GetGCost() + N.GetArcCost(N2));
             // HCost pas recalculé car toujours bon
             N2bis.calculCoutTotal(); // somme de GCost et HCost
             // Mise à jour de la famille ....
                    N2bis.Supprime_Liens_Parent();
                    N2bis.SetNoeud_Parent(N);
                  // Mise à jour des ouverts
                  L_Ouverts.Remove(N2bis);
                  this.InsertNewNodeInOpenList(N2bis);
              } // else on ne fait rien, le nouveau chemin est moins bon
           }
          else
           {
              // N2 est nouveau, MAJ et insertion dans les ouverts
              N2.SetGCost(N.GetGCost() + N.GetArcCost(N2));
              N2.CalculeHCost();
              N2.SetNoeud_Parent(N);
              N2.calculCoutTotal(); // somme de GCost et HCost
              this.InsertNewNodeInOpenList(N2);
            }
         }// else il est dans les fermés donc on ne fait rien,
    }}
```

Application : Soit le graphe suivant :



Faites tourner l'algorithme de Dijkstra manuellement pour trouver le plus court chemin entre A et E sur l'exemple précédent en montrant l'évolution de O et de F jusqu'à la solution. Voir un exemple corrigé à la fin du chapitre.

Remarques :
- L'algorithme de Dijkstra est relativement performant car il ne nécessite pas de prendre en compte tous les nœuds du graphe ou les états de l'espace d'états du problème. Cependant, il effectue une recherche "radiale" et ne tient pas compte d'informations contextuelles. Par exemple, si on cherche le plus court chemin entre Paris et Marseille et qu'on dispose des routes de la carte de France, l'algorithme explore toutes les villes au nord, à l'est et à l'ouest qui sont à une distance inférieure à la distance entre Paris et Marseille. On ne peut pas dire que ce soit très efficace relativement au problème posé !
- Dans certains cas, il existe plusieurs solutions et on cherche justement à les trouver toutes. Une variante de cet algorithme consiste alors à poursuivre la recherche jusqu'à ce qu'un autre critère de fin soit satisfait, par exemple le nombre de solutions trouvées atteint un certain nombre ou le coût du chemin excède un certain seuil.

6.2. Algorithme A*

L'A* (prononcez A étoile) est une amélioration de l'algorithme de Dijkstra lorsqu'il est possible d'exploiter des connaissances sur le problème pour rendre la recherche du plus court chemin plus efficace.
L'idée est de modifier le calcul du coût d'un chemin en associant à chaque nouveau sommet non pas le coût du chemin à partir du sommet initial, mais la somme de ce coût avec une estimation approximative dite heuristique du coût restant pour atteindre le sommet terminal. On peut démontrer que si cette estimation est un minorant du véritable coût du chemin restant à parcourir, alors la solution trouvée est optimale. Si elle ne l'est pas, la solution trouvée reste souvent proche de l'optimalité et permet un gain de temps parfois appréciable. Attention, il doit s'agir d'une estimation selon un calcul rapide qui ne doit pas obliger à faire une recherche dans l'espace d'états !
Exemple : Dans un réseau routier, si l'objectif est de partir d'une ville et d'en atteindre une autre en effectuant le moins de kilomètres possible, une estimation du chemin restant à parcourir lorsqu'on est à une ville donnée est la distance à vol d'oiseau, qui est nécessairement un minorant de la distance effective.
Reprenons l'algorithme avec les notations suivantes :
A est le sommet initial et Z le sommet terminal.
O et F sont respectivement l'ensemble des ouverts et des fermés, comme pour Dijkstra.
Pour un nœud N donné, on note g(N) le coût pour rejoindre N à partir de A, h(N) le coût de l'heuristique pour aller de N à Z. On a donc une estimation du coût total du chemin par la fonction suivante :

f(N) = g(N) + h(N)

L'A* fonctionne donc exactement comme Dijkstra, sauf qu'on remplace la fonction g(N) par la fonction f(N) pour ordonner les noeuds de la liste des Ouverts.

6.3. Trouver des heuristiques

Il n'est pas toujours évident de trouver des heuristiques à la fois efficaces pour minimiser l'espace de recherche et préservant l'optimalité de la solution. Une méthode intéressante consiste à relâcher une contrainte du problème pour qu'il devienne facile d'estimer le chemin restant à partir d'un noeud donné.
Exemple avec le jeu du taquin :
Pour passer d'un état à un autre, la contrainte est de déplacer un carré d'une case A vers une case B si A est adjacent à B horizontalement ou verticalement et si B est vide.

a) Admettons qu'on autorise un déplacement adjacent même si B n'est pas vide, comme si on pouvait passer par-dessus d'autres carrés. Il est alors facile de trouver la solution optimale et donc d'estimer le nombre de déplacements restants à partir d'un état du jeu : il est égal à la somme des distances de Manhattan entre chaque carré mal placé et sa position finale (cette distance vaut la somme des valeurs absolues des différences en x et en y, ce qui correspond à la distance parcourue à Manhattan quand on contourne les immeubles en suivant des rues orientées nord-sud ou est-ouest).
b) Admettons qu'on autorise un déplacement entre 2 cases même si celles-ci ne sont pas adjacentes, comme si on pouvait démonter le jeu et remettre directement les carrés au bon endroit. Là encore, la

solution est évidente. Elle peut être réalisée en un nombre de coups égal à la somme des carrés mal placés.

Dans les 2 cas, on constate que le relâchement de la contrainte aboutit à une heuristique qui est une estimation inférieure ou égale au nombre de mouvements restant à faire et qui est donc exploitable dans l'A*.

Une autre idée intéressante est de chercher une solution non optimale mais rapide à obtenir selon un algorithme quelconque, puis à prendre en compte le coût total obtenu comme seuil à ne pas dépasser pour g(n) dans Dijkstra, afin de limiter l'exploration.

6.4. Recherche locale et problèmes d'optimisation

Pour de nombreux problèmes, seule la solution importe, pas le chemin qui y conduit. Par exemple, le problème des 8 reines consiste à placer 8 reines sur l'échiquier sans que celles-ci soient en prise avec une autre reine. Une technique d'exploration classique de l'espace d'états consiste à tenter de placer les reines 1 par 1 en respectant les contraintes de prise jusqu'à en avoir 8. Cependant, peu importe le chemin parcouru dans l'espace d'états avec des états intermédiaires comportant moins de 8 reines, seul nous intéresse l'état final avec les 8 reines sur l'échiquier.

Principes de la recherche :

Lorsque le cheminement dans l'espace d'état n'est pas important, on peut résoudre le problème d'une autre façon, en effectuant une "recherche locale" de la solution, en partant d'un état quelconque. Par exemple, pour le problème des 8 reines, on peut placer 8 reines sur l'échiquier au hasard pour constituer un état du jeu. En général, cet état est très loin de l'état solution. Puis, on envisage le déplacement de certaines reines à d'autres positions et on compare les états obtenus avec l'état précédent. On retient alors l'état qui est le mieux évalué et on recommence. Au fur et à mesure qu'on tente des nouveaux déplacements de reine, on se rapproche de la solution et on finit généralement par la trouver.

Néanmoins, il se pose un nouveau problème : avec cette recherche locale, est-on assuré de trouver la solution ? On peut facilement montrer qu'on trouve toujours la solution si pour tout état de l'espace d'état, les actions locales envisagées à partir de celui-ci conduisent à au moins 1 autre état qui est mieux évalué (qui est donc plus proche de la solution). Si ce n'est pas le cas, il est possible de trouver la solution en procédant à des déplacements plus importants dans l'espace d'état (la recherche est moins locale), éventuellement de manière aléatoire.

Discussion :

L'avantage de la recherche locale est de limiter les besoins en espace mémoire. De plus, certaines contraintes du problème permettent parfois de trouver facilement un état proche de l'état solution, épargnant ainsi des temps de calculs importants dans la construction d'un arbre de recherche exhaustif. Le problème de la recherche locale se ramène donc à trouver l'état final en se déplaçant **localement** dans l'espace d'état et en cherchant le maximum d'une fonction d'évaluation objective, ou le minimum d'une fonction de coût (estimation de la distance à un état théorique optimal). Ceci peut être visualisé par la courbe présentée Figure 1. Si on part d'un état central proche de la solution, la pente de la fonction de coût est favorable et on trouvera la solution. Si en revanche on part d'un état trop loin, il est possible de tomber dans un minimum local. Ceci est d'autant plus dommageable que pour certains problèmes, on cherche à trouver une solution optimale et que les minimums locaux ne correspondraient qu'à des solutions quelconques mais qu'il n'est pas possible de distinguer *a priori* de la solution optimale. Par exemple, si on cherche le rangement de meubles et caisses qui prend le moins de place dans un camion, chaque état correspond à un rangement acceptable. La recherche locale consisterait par exemple à tenter des permutations sur la position des meubles. On pourrait ainsi converger vers une solution meilleure, mais rien ne garantit que ce serait la solution optimale. Ce genre de problème fait partie des **problèmes d'optimisation**, classiques en recherche opérationnelle.

Remarque 1 : La méthode dite du **recuit simulé** (simulated annealing) s'inspire des procédés de métallurgie pour obtenir les propriétés requises. On procède typiquement par plusieurs augmentations et baisses de la température (d'où le terme de "recuit"). Concrètement, il s'agit d'essayer de passer outre les minima locaux pour tomber sur le vrai minimum en effectuant de temps en temps un déplacement important dans l'espace d'état (de façon plus ou moins aléatoire) en envisageant des états dont le coût est plus important (on remonte la température) à partir desquels une nouvelle recherche locale est appliquée en suivant la pente descendante.

Remarque 2 : les **algorithmes génétiques** sont des variantes de ces algorithmes de recherche locale. La différence fondamentale est que les nouveaux états candidats sont choisis selon une combinaison d'états parents, au lieu d'un seul.

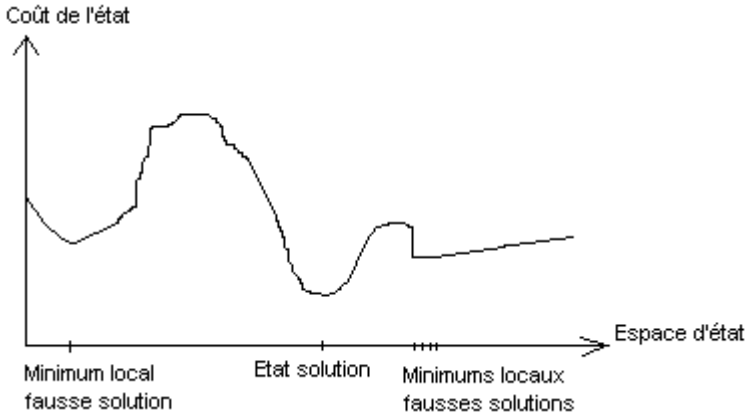
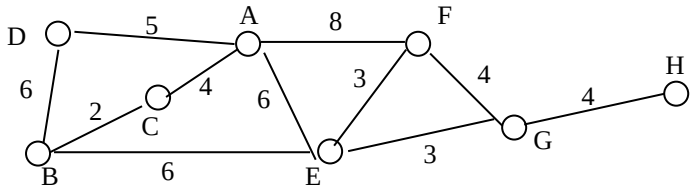


Figure 1: Problème de la recherche locale et de nombreux problèmes d'optimisation : comment trouver le minimum global de la fonction de coût par des mouvements locaux dans l'espace d'état ?

6.5. Exercices

Exercice corrigé 1 :

Appliquez l'algorithme de Dijkstra au graphe ci-dessous pour déterminer le plus court chemin entre **D** et **E** (peu importe les autres chemins). Montrez l'évolution des ensembles $\mathcal{F}$ et O qui désignent respectivement la liste des sommets "fermés" et la liste des sommets "ouverts", le tableau des distances, le tableau des prédécesseurs et l'arbre d'exploration. Vous arrêterez la simulation de l'algorithme aussitôt que le meilleur chemin entre D et E est trouvé, en précisant quelle est la condition d'arrêt requise.



Réponse :

(1) $\mathcal{F}$ = {} O ={D}
(2) $\mathcal{F}$ = {D} O ={A, B}
(3) $\mathcal{F}$ = {D, A} O ={B, C, E, F}
(4) $\mathcal{F}$ = {D, A, B} O ={C, E, F}
(5) $\mathcal{F}$ = {D, A, B, C} O ={E, F}
(6) $\mathcal{F}$ = {D, A, B, C, E}

E passe dans les fermés, c'est la condition d'arrêt de l'algorithme, le plus court chemin entre D et E est trouvé, c'est D, A, E.

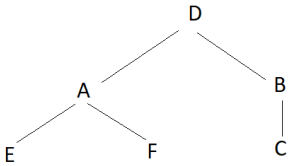
Evolution du tableau des distances :

	A	B	C	D	E	F	G	H
(1)	∞	∞	∞	0	∞	∞	∞	∞
(2)	5	6	∞	0	∞	∞	∞	∞
(3)	5	6	9	0	11	13	∞	∞
(4)	5	6	**8 (!!!)**	0	11	13	∞	∞
(5)	5	6	8	0	11	13	∞	∞
(6)	5	6	8	0	11	13	∞	∞

Evolution du tableau des prédécesseurs :

	A	B	C	D	E	F	G	H
Préd. : (1) à (3)	D	D	A	-	-	-	-	-
(4)	D	D	**B (!)**	-	A	A	-	-
(5) et (6)	D	D	B	-	A	A	-	-

Arbre d'exploration :



Exercice corrigé 2 :

L'algorithme de Dijkstra permet-il de trouver le plus court chemin si on l'applique à un graphe dans lequel on a associé à certains arcs une valeur négative ? Etayez votre propos par une démonstration ou un contre-exemple.

Réponse :

Contrexemple : il suffit de modifier le coût de l'arête FE et de lui donner la valeur négative -10. Dans ce cas, le plus court chemin serait D,A,F,E. Or, l'algorithme de Dijkstra commencerait comme ci-dessus et après que E soit passé dans les fermés, il n'y aurait plus de mise à jour possible du chemin menant à E. Donc Dijkstra ne trouverait pas le chemin D, A, F, E. CQFD, Dijkstra ne fonctionne pas si les coûts sont négatifs.

Exercice corrigé 3 :

Pour déplacer un robot de sa position actuelle à une position finale souhaitée, on construit une grille imaginaire sur laquelle le robot peut se déplacer horizontalement, verticalement ou diagonalement d'une case à la fois et on tente de trouver un chemin sur cette grille. Lorsqu'un obstacle est présent dans la case considérée, même partiellement, la case est considérée comme inaccessible. Pour un déplacement horizontal ou vertical d'une case, on associe un coût de 1 et pour un déplacement en diagonal, on associe un coût de 1,414. Pour déterminer le plus court chemin de la position courante du robot à la position recherchée, on utilise l'A*.

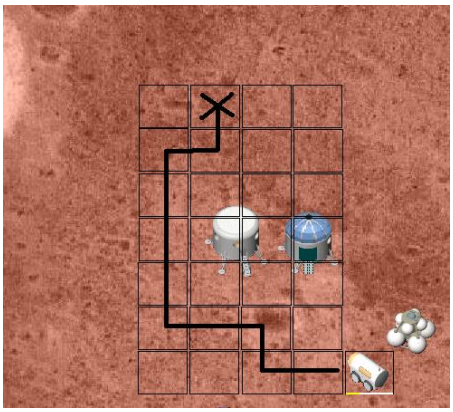


Figure 1 : Illustration du déplacement d'un robot (sans les diagonales pour cet exemple)

Question 1 : Que proposez-vous comme fonction heuristique pour minimiser les temps de calculs ? Pour rappel, une fonction heuristique d'un A* est une fonction qui permet d'estimer, de façon approximative, le chemin restant à parcourir entre une position envisagée et la position terminale recherchée. Plus l'estimation est proche de la véritable distance à parcourir tout en y étant inférieure ou égale, plus la recherche est efficace. La qualité de l'heuristique proposée sera prise en compte dans la notation. Expliquez votre choix.

Question 2 : Sur la figure 1, on montre le déplacement d'un robot sur une grille imaginaire en interdisant les déplacements en diagonal. De plus, la taille des cases est relativement importante, ce qui conduit à des trajectoires non optimales. Pourquoi de tels choix ? A quelle condition peut-on construire un graphe avec des arcs correspondant à des déplacements en diagonal ?

Réponse 1 :

On note (xf, yf) la position de la case à atteindre et (x, y) la position de la case qu'on est en train d'examiner, c'est-à-dire celle associée au nœud du graphe qu'on va passer dans les fermés dans l'algorithme de Dijkstra/A*. Appelons ce nœud N et h la fonction heuristique.

* Première possibilité simple, prendre en compte la distance à vol d'oiseau.

Dans ce cas :

h(N) = racine carrée ((xf-x)*(xf-x) + (yf-y)*(yf-y)).

* Deuxième possibilité, essayons d'affiner notre estimation, tout en préservant la contrainte d'être toujours inférieur à la véritable distance qu'il reste à parcourir. Dans le cas le plus favorable, il n'y a aucun obstacle à

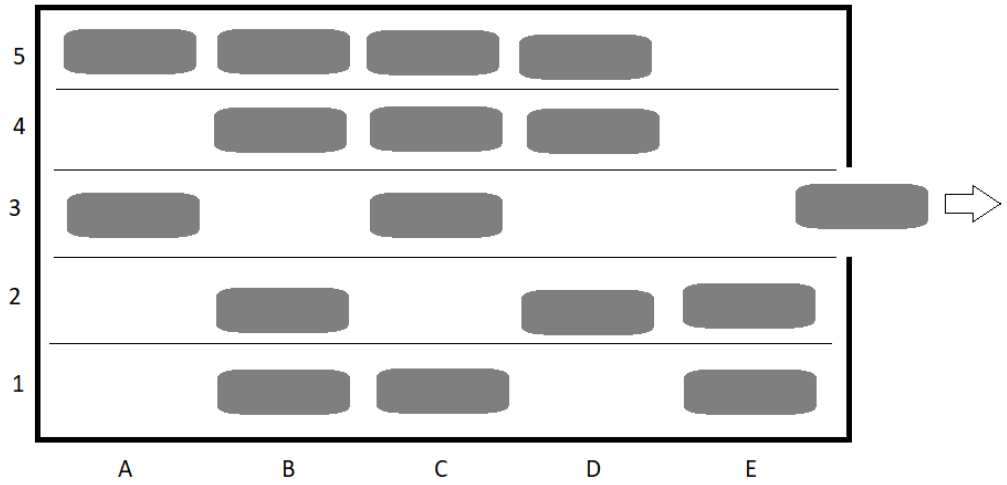
contourner et la distance à parcourir est égale à un déplacement en diagonal + un déplacement horizontal ou vertical, selon le contexte. Distinguons les 2 cas :
On note "abs" la valeur absolue.
si (abs (xf-x) > abs(yf-y)), h(N) = 1,414*abs(yf-y) + (abs(xf-x) – abs(yf-y))
sinon h(N) = 1,414*abs(xf-x) + (abs(yf-y) – abs(xf-x))

Réponse 2 :
La taille des cases correspond à la taille de l'objet à déplacer. Si on prend des cases plus petites, le risque est de trouver un chemin qui conduira l'objet à heurter des obstacles au niveau de ses parties qui dépassent de la case (de même, un humain ne choisit pas de faire passer son centre de gravité à 1 cm d'un mur car il risque de toucher le mur au niveau des jambes, des bras ou des hanches !).
En ce qui concerne les déplacements diagonaux, ce qui peut poser problème, c'est d'avoir la case de départ libre, la case située en diagonale également libre, mais une des 2 cases adjacentes occupée par un obstacle. Lors du passage diagonal, il y a donc un risque de toucher un obstacle. Pour assurer un déplacement diagonal entre une case A et une case B, la contrainte est d'avoir les 2 cases simultanément adjacentes à A et B également libres.

Exercice 4 :
Problème du voyageur de commerce
Un commerçant souhaite visiter toutes les villes d'une région en effectuant le plus court chemin dans un réseau routier dont on connaît les distances entre villes. On suppose qu'une ville est présente à chaque intersection et qu'il part obligatoirement d'une ville donnée.
a) Modélisez la résolution de ce problème en définissant toutes les modalités permettant d'effectuer une recherche efficace dans l'espace d'états, notamment comment est défini un état de l'espace d'états, comment est caractérisé l'état initial, comment est définie la fonction successeur, etc.
b) Quelle heuristique proposez-vous pour accélérer la recherche ?

Eléments de réponse :
Un état est défini par la position (la ville) du commerçant ainsi que par la liste des villes déjà visitées. En effet, il faut pouvoir faire la différence entre « commerçant dans la ville X et passé dans 3 villes » et « commerçant dans la ville X et toutes les villes ont été visitées », ce qui est un état final. Une heuristique possible est la somme des longueurs minimales des arêtes qui arrivent sur chaque ville qui reste à visiter.

Exercice 5, résolution générale de problème :
Un voiturier souhaite optimiser la place occupée sur le parking dont il a la responsabilité et n'hésite pas à le surcharger. Le parking est rectangulaire et peut être défini par 5 rangées de 5 places de voiture. Les rangées sont numérotées de 1 à 5 et les colonnes de A à E, comme indiqué ci-dessous.



Le parking étant souvent saturé, le principal problème est de sortir une voiture par une sorte de jeu de taquin où il s'agit de déplacer plusieurs voitures afin de libérer le passage pour celle qui doit sortir. On considère les hypothèses et contraintes suivantes :

- Pour sortir, une voiture doit atteindre la position E3.

- A l'exception de la voiture qu'on est venu chercher, aucune autre voiture ne doit sortir du parking. Une voiture peut facilement être déplacée de une ou plusieurs colonnes en restant sur la même rangée (déplacement vers l'avant ou vers l'arrière), à condition bien entendu qu'il n'y ait pas de voiture qui occupe les positions intermédiaires. Pour un tel déplacement, on considère qu'il faut exactement 1 minute, quel que soit le nombre de colonnes de ce déplacement.
- Une voiture peut rester sur la même colonne mais changer de rangée, c'est-à-dire monter ou descendre d'un cran. Dans ce cas, la manœuvre est plus complexe et prend 2 minutes.
- Le voiturier peut se faufiler entre les voitures pour accéder à la voiture de son choix. On considère que le temps mis pour sortir d'une voiture et accéder à une autre voiture est négligeable.
- Aucun autre déplacement n'est autorisé.

Question 1 : Modélisation

L'objectif est de minimiser le temps mis par le voiturier pour sortir du parking une voiture située à une place prédéterminée. Proposez la résolution de ce problème par une modélisation qui consiste à explorer l'espace d'états selon une technique de recherche du plus court chemin de type Dijkstra (comme vu en cours). Pour cela, vous indiquerez en particulier :

- Ce qui caractérise un état (et donc un nœud dans l'arbre de recherche).
- Comment est caractérisé l'état initial.
- Comment est caractérisé l'état final.
- Comment sont définis les états successeurs d'un état donné et le coût des transitions.

Réponse :

Un état est défini par 2 choses :

- une grille 5x5 de booléens représentant l'état de chaque place de parking (occupé / non occupé)
- la position (rangée, colonne) de la voiture à sortir

En effet, à part pour la voiture à sortir, il importe peu de savoir quelle est exactement la voiture située à une place donnée, pour 2 grilles identiques, l'effort restant pour arriver à l'état final sera le même, d'où l'utilisation de booléens pour la grille. En revanche, il faut savoir faire la différence entre 2 grilles différentes, car les états successeurs seraient différents. Enfin, il faut l'information de position de la voiture à sortir, afin de pouvoir identifier l'état final.

Etat initial : la grille correspondant aux positions initiales des voitures et la position (x,y) de la voiture à sortir.

Etat final : la voiture à sortir en position E3.

Etats successeurs : on parcourt la grille ; pour chaque case occupée, on a potentiellement des états successeurs correspondant au mouvement possible de la voiture considérée, c'est-à-dire, monter ou descendre d'une rangée, avancer ou reculer de 1 ou plusieurs cases en respectant les contraintes de l'énoncé : il faut que la case d'arrivée de la voiture existe et soit vide et, concernant le déplacement horizontal, il faut en plus que les cases intermédiaires soient également vides. Si c'est un déplacement vers le haut ou le bas, le coût de la transition est de 2 minutes et si c'est un déplacement horizontal, le coût est de 1 minute.

Question 2 : Heuristique pour A*

L'exploration de l'espace d'états peut s'avérer très longue. Pour optimiser les temps de calcul, il serait judicieux de trouver une heuristique et de passer de Dijkstra à l'A*. Que proposez-vous comme heuristique performante pour estimer le temps restant pour passer d'un état donné à l'état final ? Justifiez votre choix.

Réponse :

On peut facilement estimer un minimum du temps restant pour atteindre l'état final en s'intéressant uniquement au déplacement de la voiture à sortir sans tenir compte d'éventuelles autres voitures mal placées. Considérons la grille avec 5 colonnes et 5 lignes. Si la voiture à sortir est en x,y, sachant que la sortie est en (5,3) selon l'énoncé, il faudra au minimum valeur absolue de (y-3) fois 2 minutes pour arriver sur la bonne ligne et 1 minute pour un déplacement horizontal jusqu'à la colonne 5.

Donc si N est le nœud considéré et si on appelle h(N) l'heuristique pour ce nœud,

$h(N) = 2(|y-3|) + 1$

7. Problèmes de satisfaction de contraintes

7.1. Définition formelle

Un problème de satisfaction de contraintes est défini par un ensemble de variables X1..Xn, et un ensemble de contraintes C1..Cm. Chaque variable Xi a un domaine de définition non vide et chaque contrainte implique un sous-ensemble des variables en spécifiant les combinaisons de valeurs qui sont autorisées.
Un état du problème est défini par une affectation de valeur à certaines variables, de façon compatible avec certaines contraintes. On dit que l'affectation est **consistante** lorsqu'il y a compatibilité avec toutes les contraintes. La solution à un problème de satisfaction de contraintes est une affectation complète (chaque variable a une valeur) et consistante.
Dans certains cas, il y a plusieurs solutions au problème, et il est parfois demandé que la solution maximise une fonction d'évaluation.
Exemple de problème de satisfaction de contraintes :
Soit 7 régions A, B, C, D, E, F et G représentées sur une carte, dont certaines ont une frontière commune (AB,AD,BC,BD,CD,CE,DE,DF,EF). Soient 3 couleurs rouge, vert et bleu. Si cela est possible, proposez une couleur pour chaque région de sorte qu'il n'y ait jamais 2 régions voisines qui soient de la même couleur.
NB : dans le cas général, il existe toujours une solution pour 4 couleurs, mais pour 3, ce n'est pas certain.
Modélisation du problème :
Il y a 7 variables.
Le domaine de définition de chaque variable est {rouge,vert,bleu}
Les contraintes peuvent être définies par la proposition suivante, toujours vraie :
$\forall$ X et Y 2 régions, voisins(X,Y) $\rightarrow$ not (couleur(X)=couleur(Y))
avec voisins(X,Y) fonction booléenne qui renvoie vrai Ssi X et Y ont une frontière commune et couleur(X) qui renvoie la couleur attribuée à la région X.
Une solution au problème est par exemple : A=rouge, B=vert, C=rouge, D=bleu, E=vert, F=rouge, G=rouge
Autres problèmes :
- Problème des 8 reines : placez sur un échiquier 8 reines, telles qu'aucune d'entre elles soient en prise avec une autre.
- Soit n cours de 1 heure avec k enseignants et m salles à répartir dans la semaine, attribuez une salle pour chaque cours de sorte qu'un enseignant n'ait pas 2 cours à la fois, qu'il n'y ait pas 2 cours en même temps dans la même salle, que X ne travaille pas le lundi et Y ne travaille pas le mercredi.
- Trouvez une affectation d'un chiffre entre 0 et 9 aux variables L,U,N,E,M,A,R,S et V, pour que le résultat de l'opération suivante soit exact : LUNE + MARS = VENUS, avec comme contrainte supplémentaire que L, M et V ne soient pas égales à 0 et que chaque variable ait une valeur différente des autres.

7.2. Recherche de la solution

Le problème de satisfaction de contraintes peut être résolu de manière classique par une technique de recherche dans un espace d'état.
Etat initial : aucune affectation
Fonction successeur : liste des affectations possibles d'une variable à une valeur en préservant la consistance
Fonction test d'état final : affectation complète
Fonction de coût : peu importe, par exemple 1 à chaque nouvelle variable affectée
Il est évident qu'une solution au problème apparaît à une profondeur égale au nombre de variables à affecter. L'algorithme de parcours typiquement utilisé pour ce genre de problème est précisément un parcours en profondeur qui minimise le nombre de calculs pour peu qu'on soit efficace dans l'affectation de chaque nouvelle variable et qu'il existe une solution.
Remarque 1 : le chemin qui mène à la solution importe peu, seule l'état final nous intéresse. Une autre modélisation consiste ainsi à partir d'une affectation de toutes les variables sans respecter toutes les contraintes, puis à effectuer des changements pour tenter de trouver la solution.
Remarque 2 : Les méthodes de recherche locales sont en général très efficaces pour traiter ce genre de problème. On peut remarquer en particulier que s'il n'y a pas beaucoup de valeurs ni de variables, une recherche exhaustive peut être envisagée en tentant toutes les combinaisons d'attribution possible. Par ailleurs, il est souvent possible d'exploiter avantageusement les données du problème pour démarrer avec un état initial très proche de la solution. Par exemple, pour le problème des 8 reines qui ne doivent pas être en prise sur l'échiquier, on peut choisir de placer une reine dans chaque colonne pour former l'état initial, puis à n'envisager que des mouvements en colonne pour trouver la ou les solutions.

7.3. Discussion sur la faisabilité

Le domaine de définition de certaines variables peut être infini. L'espace d'état ayant une largeur infinie, il n'est alors pas toujours possible de trouver une solution. De même les contraintes peuvent être définies par des inégalités tels que (cos(X) + 1 < Y), ce qui rend la recherche particulièrement complexe. Lorsque les contraintes s'expriment par des équations ou inéquations linéaires, on parle de programmation linéaire.
Lorsque les contraintes sont décrites par des équations qui ne font intervenir qu'une seule variable (contrainte unaire), il suffit d'en tenir compte directement dans la fonction successeur, ou dans la recherche locale.
Lorsque les contraintes sont décrites par des équations binaires, on représente généralement celles-ci à l'aide d'un graphe dont les nœuds sont les variables et les arcs correspondent précisément à la relation binaire qui lie 2 variables.
De nombreux travaux en recherche opérationnelle ont été réalisés pour proposer des solutions générales à des catégories de problème de satisfaction de contraintes. Ces catégories de problème se distinguent en fonction du type des variables et du type des contraintes (unaires, binaires, équations linéaires, non linéaires, etc.).

7.4. Exercices

Exercice corrigé 1 :

Voici ci-dessous une énigme classique trouvée dans une revue, dans laquelle il faut déterminer les 5 lettres d'un mot. On peut noter que toutes les lettres de ce mot se trouvent dans les autres mots de la grille.

Question 1 : On souhaite résoudre cette énigme à l'aide d'un programme informatique. A quelle catégorie de problèmes appartient cette énigme ? Justifiez votre réponse en donnant d'autres exemples de problème du même style.

Question 2 : Proposez une modélisation informatique du problème en effectuant une recherche dans l'espace d'états selon une technique classique de résolution générale de problèmes. Répondez notamment aux questions suivantes :
- Quelle structure de données proposez-vous pour modéliser un état du problème ?
- Quel est l'état initial ?
- Comment est caractérisé l'état final ?
- Donnez les grandes lignes de la fonction "successeur" permettant de trouver les états suivants d'un état donné.
- Quelles sont les grandes lignes de la fonction permettant de vérifier qu'un état est consistant par rapport aux contraintes du problème ?
- Au niveau de la fonction successeur, quelle stratégie proposez-vous pour déterminer le choix de la lettre à placer ?

Réponse 1 : Dans la catégorie des problèmes de satisfactions de contraintes, comme Sudoku, 8 reines, …

Réponse 2 *: Quelle structure de données proposez-vous pour modéliser un état du problème ?*
Un état est typiquement déterminé par des lettres placées dans le mot secret, entre 0 et 5 lettres.
structure typique, un tableau de 5 caractères :

```
char[5] t;          // "?" pour les cases non remplies
```

Le tableau des contraintes pourra être stocké en variable globale initialisée avant la recherche dans l'espace d'état. Structures utilisées : 1 tableau de 5x10 cases de string pour les mots et dans un tableau de 10 cases d'entiers pour les scores.

- *Quel est l'état initial ?* un tableau rempli de 5 "?"
- *Comment est caractérisé l'état final ?* par un tableau rempli de 5 lettres différentes de "?"
- *Donnez les grandes lignes de la fonction "successeur" permettant de trouver les états suivants d'un état donné.*
Pour un état N donné, comprenant k cases non encore déterminées, on procède en 2 étapes :
a) Choix de la nouvelle case à placer (un indice entre 0 et 4 parmi les k indices encore possibles)
b) On regarde les lettres possibles pour la case choisie : elle doit faire partie de la liste des lettres contenues dans le tableau des contraintes dans la même colonne (même indice). Cette liste fournit autant d'états successeurs potentiels. Cependant, il faut vérifier qu'en ajoutant une lettre, les contraintes sont toujours satisfaites. La liste des états successeurs est donc donnée par la liste des états formés en ajoutant une nouvelle lettre dans la colonne choisie tout en respectant les contraintes.

- *Quelles sont les grandes lignes de la fonction permettant de vérifier qu'un état est consistant par rapport aux contraintes du problème ?*
On prend chaque mot du tableau des contraintes et on le compare au mot associé à l'état dont on veut vérifier la consistance. Il faut compter le nombre de lettres qui sont les mêmes et comparer ce chiffre au chiffre correspondant du tableau des scores. Si c'est inférieur ou égal et qu'il reste suffisamment de cases pour atteindre le chiffre des socres, c'est consistant, sinon ça n'est pas consistant. Un état est consistant si et seulement si la consistance est validée pour tous les mots du tableau des contraintes.

- *Au niveau de la fonction successeur, quelle stratégie proposez-vous pour déterminer le choix de la lettre à placer ?*
La stratégie la plus simple consiste à commencer à l'indice 1 et de continuer de 1 en 1 jusqu'à l'indice 5. Par exemple si on a déjà placé les n premières lettres du mot à trouver, on s'intéressera à la lettre à placer à l'indice n+1. Une stratégie plus pertinente consiste à choisir la colonne (l'indice) la plus contrainte, parmi les colonnes non encore choisies. Pour cette application, on peut déterminer le nombre de lettres différentes de la colonne et choisir comme colonne celle où il y a le moins de choix (moins de choix, donc + de contraintes).

Exercice corrigé 2 :
On considère le problème de l'affectation d'un chiffre entre 0 et 9 aux variables D,E,U,X,S,P,T,N et F, pour que le résultat de l'opération suivante soit exact : DEUX + SEPT = NEUF, avec comme contrainte que chaque variable ait une valeur différente des autres.
Donnez les grandes lignes d'une méthode de résolution de ce problème par un programme informatique en vous inspirant de techniques connues pour résoudre des problèmes similaires.

Réponse :
Il s'agit d'un problème classique de satisfaction de contraintes, comme le problème des 8 reines ou du sudoku (voir cours).
Ce type de problème est caractérisé par un ensemble de variables dont il faut déterminer la valeur et un ensemble de contraintes à respecter.
Les variables sont les 9 entiers D,E,U,X,S,P,T,N et F.
La contrainte principale peut s'écrire de la manière suivante :
(1000D + 100E + 10U + X) + (1000S + 100E + 10P + T) = (1000N + 100E + 10U + F)
A laquelle s'ajoute les contraintes de non égalité des variables et la contrainte d'intervalle de valeurs entre 0 et 9 pour chacune d'elles.
On peut également ajouter des contraintes qui découlent de l'analyse du problème.
Par exemple, on ne peut avoir T=0 (sinon X=F) ni X=0 (sinon T=F) ni F=0 (sinon X=T=0 ou X=T=5).
De plus, E vaut 0 ou 9 pour pouvoir avoir E + E + retenue possible 0 ou 1 = E

La résolution du problème (voir cours) consiste à suivre les étapes algorithmiques suivantes, qui s'apparentent à une recherche d'un plus court chemin dans l'espace d'états selon un parcours en profondeur d'abord.
(0) Aucune variable n'est affectée
(1) Choisir la variable la plus contrainte parmi celles qui ne sont pas encore affectées
(2) Affecter une valeur à cette variable parmi celles qui n'ont pas encore été essayées (chemins ouverts de l'espace d'états) et qui permettent de respecter les contraintes. S'il ne reste plus de valeur possible, aucune solution, fin.
(3) Si toutes les variables sont affectées, fin, on a trouvé
(4) Aller en (1)

NB : la variable la plus contrainte est celle qui comporte le moins de chiffres possibles. Par exemple, après l'analyse proposée, c'est E puisqu'elle ne peut prendre comme valeur que 0 ou 9, ensuite T, X et F.

Exercice 3 :
Sudoku
Proposez une modélisation de la recherche de solution pour le Sudoku. Spécifiez en particulier l'espace d'états, l'état initial et la fonction successeur. Argumentez vos choix.

Exercice 4 :
Mots croisés indexés
Dans certains mots-croisés, la liste des mots utilisés pour la grille est fournie et l'objectif consiste à retrouver la place de chacun. Proposez une modélisation de la recherche de solution pour ce jeu. Spécifiez en particulier l'espace d'états, l'état initial et la fonction successeur.

8. Algorithmes d'exploration pour les jeux avec adversaire

8.1. Algorithme Minimax

Nous considérons ici les jeux à 2 joueurs, chacun étant l'adversaire de l'autre. Nous appellerons le premier joueur Max et l'adversaire Min. Comme dans le cas de la résolution classique de problèmes, il existe un espace d'états et l'objectif est d'effectuer la recherche d'un chemin conduisant à une solution de gain ou le cas échéant au meilleur état possible. Nous disposons donc :

- D'un état initial du jeu
- D'une fonction successeur permettant de trouver tous les états accessibles (donc tous les coups possibles) à partir d'un état donné.
- D'une fonction booléenne qui détermine si un état est terminal pour ce jeu ou pas.
- D'une fonction d'évaluation qui associe une valeur à des états terminaux.

L'état initial, les états successeurs et les transitions associées (les coups jouables) définissent l'arbre du jeu.
Contrairement à une recherche classique, Min tente d'empêcher Max de trouver la solution gagnante. Max doit donc appliquer une stratégie obligeant Min à être inefficace, pour rester sur un des chemins gagnants.
Lorsqu'un jeu est intéressant, il n'est pas possible d'envisager tous les coups possibles pour déterminer une solution gagnante. Les noeuds terminaux de l'arbre d'exploration correspondent donc à des états intermédiaires de la partie. La fonction d'évaluation va donc définir la qualité d'un état du jeu par rapport à un gain hypothétique de la partie dans le futur.
Mettons que la recherche examine n coups possibles pour le 1er coup joué par Max numérotés de a_1 à a_n, puis pour le coup a_1, n' coups joués par Min numérotés de b_1 à $b_{n'}$, puis pour le coup b_1, encore n'' coups joués par Max numérotés de c_1 à $c_{n''}$. L'objectif de Max est de jouer le coup qui maximise la fonction d'évaluation. Pour la branche a_1, b_1, il va donc retenir le maximum des valeurs associées aux coups c_1 à $c_{n''}$. Appelons ce maximum m_1. Si Min joue de façon optimale, et s'il dispose d'une fonction d'évaluation similaire, il peut estimer qu'en jouant b_1 après a_1, Max risque de jouer un coup de valeur m_1. S'il joue b_2, Max risque de jouer le coup l'amenant à un autre maximum, appelons le m_2 et ainsi de suite jusqu'au maximum $m_{n'}$ pour le coup $b_{n'}$. Pour jouer de façon optimale, Min doit donc choisir le coup qui conduit au plus petit maximum parmi m_1... $m_{n'}$, c'est donc le minimum des maximums du niveau en dessous. Revenons maintenant au n premiers coups de max. Pour chacun de ces coups, on peut supposer que Min jouera ensuite le coup qui conduit à la valeur minimum des maximums du niveau en dessous. Max doit donc choisir comme premier coup celui qui oblige Min à obtenir le plus grand des minimums. On calcule donc le maximum des minimums.
Ces explications sont illustrées sur l'exemple présenté Figure 2. Cet algorithme peut être généralisé à un nombre arbitraire de niveaux. Après évaluation au niveau le plus bas, on remonte donc alternativement à chaque noeud les maximums, puis au niveau au-dessus les minimums, et ainsi de suite jusqu'au 1er niveau. Le meilleur coup est celui qui correspond précisément au maximum du 1er niveau.
Cet algorithme s'appelle Minimax.

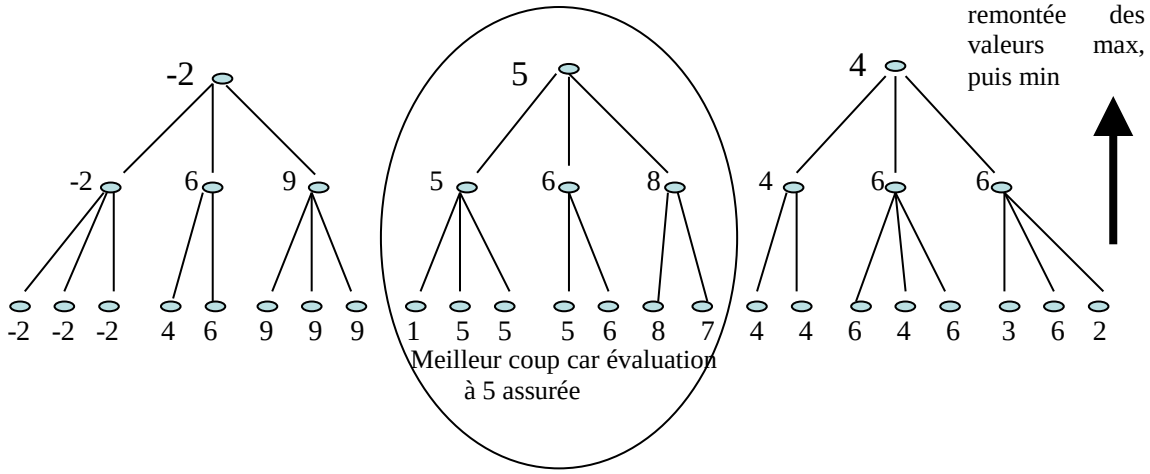


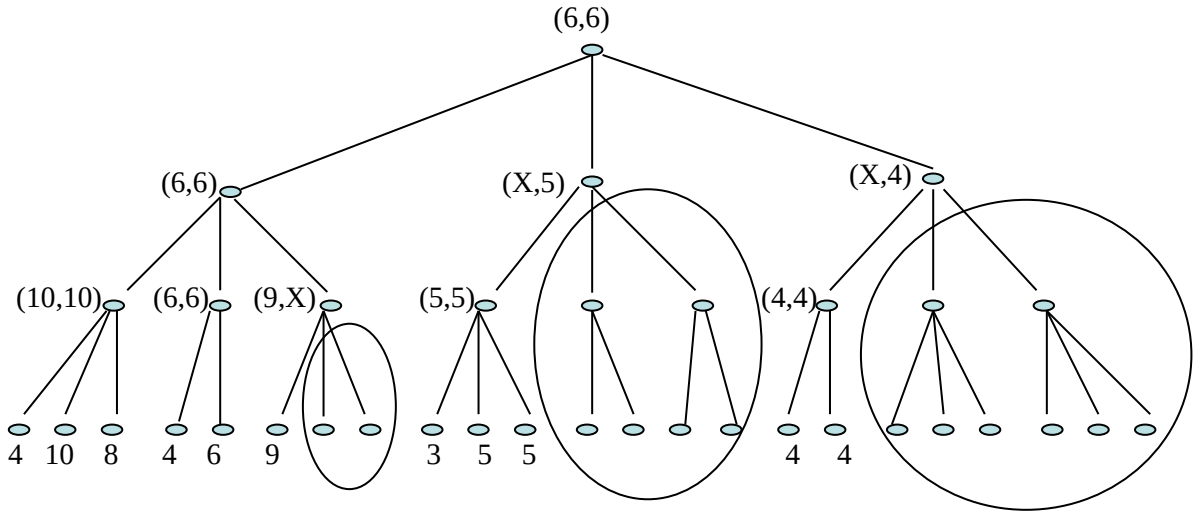
Figure 2: Arbre d'exploration avec remontée de l'évaluation par l'algorithme Minimax

8.2. Alpha Beta

L'algorithme Minimax a une complexité exponentielle : pour une profondeur donnée p, avec m coups possibles en moyenne à chaque niveau, le nombre d'états envisagés est de l'ordre de m^p. Pour des jeux comme les échecs, avec un ordinateur de performance moyenne, cela réduit la recherche à une profondeur de 3 ou 4 en milieu de partie pour quelques secondes de calcul.
Heureusement, il n'est pas nécessaire d'évaluer tous les états du jeu. Il est souvent possible d'élaguer rapidement certaines branches de l'arbre d'exploration.
Prenons un exemple en profondeur 2.
Minimax = max(min(3,12,8), min(2,x,y), min(14,5,2)) avec x et y pas encore calculées.
= max(3, min(2,x,y), 2)
= max(3, z, 2) avec $z \leq 2$
= 3
Ainsi, peu importe les évaluations x et y, le résultat ne dépend pas de ces valeurs.
Le principe général de l'algorithme alpha-beta est le suivant :
Considérons une valeur k d'un noeud N de l'arbre à une profondeur p donnée correspondant à un coup de Max. Si on a déjà fait remonter une valeur k' à un noeud de profondeur p'<p et que k' est supérieur à k, il est inutile de développer les branches soeurs de N, car Min choisira un noeud de valeur k ou moins, donc on ne peut espérer dépasser k'. Un raisonnement symétrique peut être appliqué pour un noeud correspondant à un coup de Min.
Dans cet algorithme, les valeurs α et β sont respectivement les plus fortes valeurs trouvées jusqu'ici pour Max et les plus faibles valeurs trouvées jusqu'ici pour Min. Elles sont mémorisées au fur et à mesure du parcours de l'arbre de façon récursive en profondeur d'abord.
Algorithme :

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if α ≥ β then
                break (* β cut-off *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if α ≥ β then
                break (* α cut-off *)
        return value
(* Initial call *)
alphabeta(origin, depth, −∞, +∞, TRUE)
```

Un exemple de parcours alpha-beta est proposé. Les valeurs de α et β sont indiquées entre parenthèses. 3 parties de l'arbre ne sont pas explorées, d'où un gain de temps substantiel. En effet, lors du parcours de gauche à droite, on détermine d'abord que 6 est un premier minimum et que dans la branche d'à côté, on ne peut faire moins que le 9 déjà trouvé, donc inutile d'explorer la suite. On fait donc remonter le 6 tout en haut qui devient le premier maximum connu à la racine. En redescendant au milieu, on découvre un maximum de 5. C'est inférieur au 6 déjà trouvé, donc on ne fera pas mieux, inutile d'explorer le reste de la branche du milieu. Un raisonnement analogue conduit à l'élimination de plusieurs branches dans la partie droite de l'arbre.


(6,6)
(6,6)
(X,5)
(X,4)
(10,10)
(6,6)
(9,X)
(5,5)
(4,4)
4 10 8 4 6 9 3 5 5 4 4

9. Traitement du langage naturel

9.1. Caractéristiques du langage humain

Le langage humain est défini par :

- Vocabulaire : ensemble de mots autorisés (éventuellement avec un alphabet dans le langage écrit)

L'ensemble des mots autorisés est très important (dictionnaire). En pratique, la liste des mots utilisés est relativement faible.

- Syntaxe = grammaire et conjugaison : règles pour la production de phrases correctes

Beaucoup de règles, mais beaucoup de libertés et au final bien plus complexe que les langages informatiques :
Déclinaisons en fonction de paramètres complexes (COD …)
Sujet verbe et complément pas toujours dans le même ordre …
« Vos beaux yeux, belle Marquise, me font mourir d'amour. »
« Vos beaux yeux me font mourir d'amour, belle Marquise. »

- Sémantique : sens des phrases produites relativement aux connaissances universelles, aux lois de l'univers, à l'expérience commune et aux attendus socio-culturels.
  Exemple d'erreur sémantique : « Le chien a mangé l'immeuble » ???
  Erreur plus subtile, usage inapproprié d'un mot : « Il a scié le pain. »

- Pragmatique : pertinence et cohérence de ce qui est dit relativement au contexte et à l'enchainement des idées (« Est-ce que tu veux une bière ? C'est possible, mais j'ai oublié mon parapluie. »)

Particularités du langage humain :

- Fautes de syntaxe fréquentes et relativement tolérées !

« Si je serais venu, y aurait pas eu de problème. »

- Ambiguïtés :

« Il regarde l'individu avec des jumelles. » « Prends la porte ! »

- S'inscrit dans un contexte qui apporte des informations essentielles :

« Prends cette chaise. » (pointée par le doigt, ou celle qui est la plus proche)

- Informations implicites :

« Va le chercher ! » (c'est à 10 km, donc il faut y aller en voiture)

- Nombre quasi infini de façons de dire la même chose

Dégage ! Sors d'ici ! Pars sur le champ ! Je te prie de bien vouloir quitter ce lieu au plus vite !

- Erreurs sémantiques fréquentes, mais permettant malgré tout une bonne compréhension …

« Il a fait une erreur d'inattention, ça arrive à tout le monde. »

- Possibilité d'ajout de nouveaux mots (notamment les noms propres)

- Validation sémantique à l'aide de :

Liste de connaissances historiques, géographiques, artistiques, économiques, physiques, biologiques …
Liste de connaissances sur les objets de l'univers humain (montre, voiture, stylo …)
Liste de connaissances sur les règles socio-culturelles
Liste de connaissances complexes, abstraites, implicites
+ Capacité à inférer / déduire / projeter en exploitant ces connaissances
⇨ Problème très compliqué à résoudre algorithmiquement

Exemple :
« Hier, j'ai un vu un chien planer quelques secondes au-dessus de ma maison. »
Quelles informations sont exploitées par l'humain pour invalider cette information ?

9.2. Wordnet

Wordnet : base de données lexicale pour l'Anglais, outil de structuration des connaissances, ontologie.

Liste de relations entre les noms communs :

- hypernyms: Y is a hypernym of X if every X is a (kind of) Y (canine is a hypernym of dog)
- hyponyms: Y is a hyponym of X if every Y is a (kind of) X (dog is a hyponym of canine)
- coordinate terms: Y is a coordinate term of X if X and Y share a hypernym (wolf is a coordinate term of dog, and dog is a coordinate term of wolf)
- meronym: Y is a meronym of X if Y is a part of X (window is a meronym of building)
- holonym: Y is a holonym of X if X is a part of Y (building is a holonym of window)

Liste de relations entre les verbes :

- hypernym: the verb Y is a hypernym of the verb X if the activity X is a (kind of) Y (to perceive is an hypernym of to listen)
- troponym: the verb Y is a troponym of the verb X if the activity Y is doing X in some manner (to lisp is a troponym of to talk)
- entailment: the verb Y is entailed by X if by doing X you must be doing Y (to sleep is entailed by to snore)
- coordinate terms: those verbs sharing a common hypernym (to lisp and to yell)

Cette structuration permet de répondre à des questions du type : Is_a(shark, vertebrate) ? Oui, car vertebrate est un hypernym de shark.

9.3. Analyse de textes pour répondre à des questions

Problème : on dispose de nombreux articles écrits en langage naturel contenant une multitude d'informations sémantiques. Comment exploiter ces articles pour répondre à des questions relatives à leur contenu ?

Si l'information est structurée dans des bases de données, par exemple Wordnet, le problème peut être résolu de manière simple à l'aide d'une requête, mais si l'information est non structurée le problème est très difficile. Il est possible d'exploiter des graphes sémantiques, avec de multiples relations pour relier les connaissances entre elles, mais cette approche est particulièrement fastidieuse, car il faudrait des centaines de relations différentes. En pratique, les techniques actuelles les plus performantes reposent sur des associations statistiques des groupes de mots, sans prendre en compte leur sens, ou de manière très simplifiée, ce qui limite leur capacité à répondre de manière pertinente.

Voici un texte où les groupements de mots ne suffisent pas à répondre à la question. :
Texte initial : « Un jour, alors qu'il marchait dans son quartier favori de Ulm, il décida de prendre une photo de l'avenue principale et de l'envoyer à Albert Einstein, afin que ce dernier se souvienne des beautés de sa ville natale. »
Question : « Où est né Albert Einstein ? »
Réponse attendue : « Ulm »

Difficulté importante complémentaire : comment prendre en compte l'enchainement des questions ?
Exemple : « Où est né Albert Einstein ? … Quel est le prénom de **son** père ? ».

- ⇨ La plupart des chatbot ne sont pas capables de suivre une conversation, car chaque nouvelle phrase est traitée indépendamment de la précédente.

9.4. Exemple d'outil TALN en ligne utilisé en question réponse

AllenNLP
http://demo.allennlp.org/machine-comprehension

L'application fonctionne plutôt bien pour des questions simples. Elle fonctionne par probabilité d'association de mots et d'expression trouvés à la fois dans la question et dans le texte. Ces probabilités sont obtenues après apprentissage à partir de nombreux exemples. L'application est incapable de répondre à « Was the orbiter the only reusable part of the space shuttle? » ou de répondre à des questions simples concernant le spatial.

De tels outils sont intéressants pour créer automatiquement des chatbots qui vont exploiter des textes existants pour répondre à des questions relativement standards des utilisateurs. Ces outils restent cependant très limités pour répondre à des questions plus générales et ne constituent pas une option intéressante pour passer le test de Turing.

10. Références

La bible de l'I.A. :
Artificial Intelligence : A Modern Approach, Stuart Russel et Peter Norvig, Prentice Hall Series in Artificial Intelligence, édition renouvelée régulièrement.