Chapitre 1: Introduction à l'intelligence artificielle

L'intelligence artificielle, de quoi s'agit-il ?

Je dois avouer que j'ai eu un peu de mal à écrire ce premier chapitre car il n'existe pas vraiment de consensus sur la définition du terme "intelligence artificielle"(il n'y en a même pas sur le terme "intelligence" !!). Commençons donc par citer quelques définitions (ou tentatives de définitions) que l'on peut trouver dans la littérature :

> "l'étude des facultés mentales à l'aide des modèles de type calculatoires" (Charniak et McDermott, 1985)
>
> "conception d'agents intelligents" (Poole et al., 1998)
>
> "discipline étudiant la possibilité de faire exécuter par l'ordinateur des tâches pour lesquelles l'homme est aujourd'hui meilleur que la machine" (Rich et Knight, 1990)
>
> "l'automatisation des activités associées au raisonnement humain, telles que la décision, la résolution de problèmes, l'apprentissage, ..." (Bellman, 1978)
>
> "l'étude des mécanismes permettant à un agent de perçevoir, raisonner, et agir" (Winston, 1992)
>
> "l'études des entités ayant un comportement intelligent" (Nilsson, 1998)

J'ajoute également quelques des définitions que mes étudiants de l'an dernier ont proposées parce qu'elles me semblent toutes intéressantes :

> "raisonner d'une manière autonome et également s'adapter aux changements de l'environnement... construction des machines qui ressemblent à l'être humain (vision, compréhension,...)"
>
> "la construction de programmes informatiques qui s'adonnent à des tâches qui sont, pour l'instant, accomplies de façon plus satisfaisante par des êtres humains car elles demandent des processus mentaux de haut niveau tels que la mémoire et les sentiments"

> "l'intelligence, c' est la capacité à comprendre les trucs... L' intelligence artificielle c'est si le logiciel apprend tout seul"
>
> "l'intelligence artificielle est un ensemble de techniques visant à tenter d'approcher le raisonnement humain"

Comme vous pouvez remarquer, ces définitions s'accordent sur le fait que l'objectif de l'IA est de créer des systèmes intelligents, mais elles diffèrent significativement dans leur façon de définir l'intelligence. Certaines se focalisent sur le comportement du système, tandis que d'autres considèrent que c'est le fonctionnement interne (le raisonnement) du système qui importe. Une deuxième distinction peut être faite entre celles qui définissent l'intelligence à partir de l'être humain et celles qui ne font pas référence aux humains mais à un standard de rationalité plus général. On peut donc décliner quatre façons de voir l'intelligence artificielle :

- *créer des systèmes qui se comportent comme les êtres humains* – cette définition opérationelle de l'IA fut promue par Alan Turing, qui introduisit son fameux "test de Turing" selon lequel une machine est considérée comme intelligente si elle peut converser de telle manière que les interrogateurs (humains) ne peuvent la distinguer d'un être humain [1].
- *créer des systèmes qui pensent comme des êtres humains* – si l'on adhère à cette deuxième définition, cela implique que l'IA est une science expérimentale, car il faut comprendre au préalable la façon dont pensent les humains (sinon, comment savoir si une machine pense comme un homme ?) et ensuite évaluer les systèmes par rapport à leurs similarités avec le raisonnement humain.
- *créer des systèmes qui pensent rationnellement* – selon cette définition, les systèmes doivent raisonner d'une manière rationnelle, c'est à dire en suivant les lois de la logique. Cette approche peut être critiquée car il semble que certaines capacités (la perception, par exemple) ne sont pas facilement exprimables en logique. De plus, ce standard de rationalité ne peut pas être atteint en pratique car la technologie actuelle ne permet pas de réaliser des calculs aussi complexes.
- *créer des systèmes qui possèdent des comportements rationnels* – cette dernière définition de l'IA concerne le dévelopement des agents qui agissent pour mieux satisfaire leurs objectifs. On remarque que cette définition est plus générale que la précédente car raisonner logiquement peut être une façon d'agir rationnellement mais n'est pas la seule (par

1. Des versions plus élaborées du test de Turing ont été introduites par la suite pour tester également les capacités de perception et de l'action physique.

exemple, le réflexe de retirer sa main d'un objet brûlant est rationnel mais n'est pas le résultat d'une inférence logique).

Ajoutons que dans la réalité, ces distinctions n'ont pas forcément une influence aussi grande que l'on pourrait imaginer sur la façon dont la recherche en lA est menée. Les chercheurs n'ont pas tous un avis très précis sur ce que doit être l'objectif ultime de l'IA, mais trouvent tout simplement que les questions soulevées par ce domaine sont intéréssantes et méritent d'‘être étudiées.

Une courte histoire de l'IA

Voici quelques étapes importantes dans l'histoire de l'IA :

Gestation de l'IA (1943-1955) Pendant cette période furent menés les premiers travaux qui peuvent être considerés comme les débuts de l'intelligence artificielle (même si le terme n'existait pas encore). On peut citer les travaux de McCulloch et Pitts qui ont introduit en 1943 un modèle de neurones artificiels. Quelques années après, Hebb proposa une règle pour modifier des connections entres neurones, et Minsky et Edmonds construisirent le premier réseau de neurones. Ce fut aussi durant cette période que Turing publia son fameux article dans lequel introduit le test de Turing.

Naissance d'IA (1956) C'est durant cette année qu'un petit groupe d'informaticiens intéréssés par l'étude de l'intelligence se réunirent pour une conférence sur ce thème. Cette conférence dura deux mois (!), et permit de poser les fondements de l'intelligence artificelle (nom qui fut choisi à l'issue de cette conférence)

Espoirs grandissants (1952-1969) Ce fut une période très active pour le jeune domaine de l'IA. Un grande nombre de programmes furent développés pour résoudre des problèmes d'une grande diversité. Les programmes Logic Theorist (par Newell et Simon) et Geometry Theorem Prover (Gelernter) furent en mesure de prouver certains théorèmes mathématiques (tous déjà connus, mais en trouvant parfois une preuve plus élégante). Le General Problem Solver de Newell et Simon réussissait quant à lui à résoudre des puzzles simples avec un raisonnement semblable au raisonnement humain. Samuel créa un programme jouant (à un niveau moyen) aux dames. Des étudiants de Minsky travaillèrent sur les petits problèmes ("microworlds") tels que les problèmes d'analogie (probèmes du même type que ceux des tests de QI), donnant naissance au programme ANALOGY, ou encore les manipulations de cubes (le

fameux "blocks world") avec l'idée d'augmenter la complexité petit à petit pour développer des agents intelligents. McCarthy publia un article devenu célèbre dans lequel il traite des programmes qui ont du sens commun. La recherche sur les réseaux de neurones fut également poursuivie. Ce fut aussi l'époque du Shakey, le premier robot à être capable de raisonner sur ses propres actions.

Premières Déceptions (1966-1973) Il devint durant ces années de plus en plus évident que les prédictions faites par les chercheurs en lA avaient été beaucoup trop optimistes. Ce fut le cas par exemple pour la traduction automatique. Les chercheurs n'avaient compté que 5 ans pour réaliser un traducteur automatique, mais se sont vite rendu compte que leur approche purement syntaxique n'étaient pas suffisante (pour bien traduire un texte, il faut d'abord le comprendre). Cet échec a provoqué l'annulation en 1966 de tout le financement du gouvernement américain pour les projets de traduction automatique. De grandes déceptions se produisirent également lorsque les chercheurs en IA essayèrent d'appliquer leurs algorithmes aux problèmes de grande taille, et découvrirent alors qu'ils ne fonctionnaient pas, par manque de mémoire et de puissance de calcul. Ce fut une des critiques adressée à l'IA dans le rapport de Lighthill de 1973, qui provoqua l'arrêt du financement de la quasi-totalité des projets en IA de Grande Bretagne. Et comme si cela ne suffisait pas, Minsky et Papert prouvèrent dans leur livre "Perceptrons" de 1969 que les réseaux de neurones de l'époque ne pouvaient pas calculer certaines fonctions pourtant très simples [2], ce qui mit en cause toute la recherche en apprentissage automatique, entraînant une crise dans cette branche de l'IA.

Systèmes Experts (1969-1979) Le premier système expert, appelé DENDRAL, fut créé en 1969 pour la tâche spécialisée consistant à déterminer la structure moléculaire d'une molécule étant donnés sa formule et les résultats de sa spectrométrie de masse. DENDRAL, comme toutes les systèmes experts, est basé sur un grand nombre de règles heuristiques (nous reviendrons sur ce terme en détail dans la suite du cours) élaborées par des experts humains. Après le succès du DENDRAL, d'autres systèmes d'experts furent créés, notamment le système MYCIN, qui réalisait un diagnostic des infections sanguines. Avec 450 règles, MYCIN réussissait à diagnostiquer à un niveau proches des experts humains et considèrablement meilleur que celui les jeunes médécins.

L'IA dans l'Industrie (1980-présent) Au début des années 80, l'enter-

2. Par exemple, ils démontrèrent qu'il n'existait pas de réseau de neurones capable de distinguer deux nombres écrits en binaire.

prise DEC commença à utiliser un système expert d'aide à la configuration de systèmes informatiques, ce qui leur permit d'économiser des dizaines de millions de dollars chaque année. Beaucoup de grandes enterprises commencèrent alors à s'intéresser à l'IA et à former leur propres équipes de recherche. Les Etats-Unis et le Japon financèrent de gros projets en IA, et la Grande Bretagne relança son programme de financement.

Le retour des réseaux de neurones (1986-présent) Au milieu des années 80, quatre groupes de chercheurs ont découvrirent indépendamment la règle d'apprentissage "back-propagation" qui permit le dévelopement de réseaux de neurones capables d'apprendre des fonctions très complexes (curieusement, cette règle avait déjà été proposée en 1969, mais n'avait eu aucun écho dans la communauté scientifique). Depuis, l'apprentissage automatique est devenu l'un des domaines les plus actifs de l'IA, et a été appliqué avec succès à de nombreux problèmes pratiques (comme par exemple la fouille de données).

L'IA Moderne (1987-présent L'intelligence artificielle est devenue au fil du temps une matière scientifique de plus en plus rigoureuse et formelle. La plupart des approches étudiées aujourd'hui sont basées sur des théories mathématiques ou des études expérimentales plutôt que sur l'intuition, et sont appliquées plus souvent aux problèmes issus du monde réel.

Les sous-domaines de l'IA

On s'en serait douté, créer des agents intelligents n'est pas si simple. Pour cette raison, l'IA s'est divisée en de nombreuses sous-disciplines qui essaient chacune de traiter une partie du problème. Voici les principales :

Répresentation des connaissances et Raisonnement Automatique Comme son nom le suggère, cette branche de l'IA traite le problème de la répresentation des connaissances (qui peuvent être incomplètes, incertaines, ou incohérentes) et de la mise en oeuvre du raisonnement.

Résolution de problèmes généraux L'objectif est de créer des algorithmes généraux pour résoudre des problèmes concrets.

Traitement du langage naturel Ce sous-domaine vise à la compréhension, la traduction, ou la production du langage (écrit ou parlé).

Vision artificielle Le but de cette discipline est de permettre aux ordinateurs de comprendre les images et la vidéo (par exemple, de reconnaître des visages ou des chiffres).

Robotique Cette discipline vise à réaliser des agents physiques qui peuvent agir dans le monde (pour voir les robots humanoïdes les plus avancés aujourdhui, allez sur le site `http://www.world.honda.com/ASIMO/`).

Apprentissage automatique Dans cette branche de l'IA, on essaie de concevoir des programmes qui peuvent s'auto-modifier en fonction de leur expérience.

Il existe bien entendu des liens très forts entre ces sous-domaines. Par exemple, les langages developpés dans la réprésentation des connaissances peuvent servir de base à des systèmes experts. Ou encore, beaucoup d'algorithmes pour la reconnaissance des formes sont developpés en utilisant des méthodes d'apprentissage.

Il y a aussi de forts liens entre l'IA et d'autres domaines tels que la philosophie, la psychologie, les neurosciences, les sciences cognitives, la linguistique, et l'économie.

L'état de l'art IA

Avant de clore ce chapitre, on peut citer quelques exemples qui illustrent l'état de l'art aujourd'hui :

Jeux En 1997, Deep Blue devient le premier programme à battre un champion du monde d'échecs en titre, ce qui fit sensation. Aujourd'hui, des programmes informatiques peuvent aussi jouer à un niveau expert aux dames, au backgammon, ou encore au bridge. En revanche, le jeu de go s'avère être très résistant, et aucun programme informatique ne dépasse aujourd'hui le niveau d'un joueur de club moyen.

Vision artificielle Le système ALVINN a conduit une voiture à travers des Etats-Unis pendant plus de 4000 kilomètres (avec un peu d'aide pour les moments difficiles comme les sorties d'autoroutes). On peut citer également des programmes de reconnaissance d'écriture qui arrivent à catégoriser les chiffres manuscrits avec moins d'un pourcent d'erreur (voir le site `http://yann.lecun.com/exdb/lenet/index.html` pour voir quelques démonstrations d'un tel programme).

Planification et Ordonnancement Un système de planification et d'ordonnancement a contrôlé sans la moindre intervention humaine une navette spaciale pendant 2 jours, planifiant ses actions, détectant des problèmes, et modifiant ses trajectoires au besoin.

Systèmes Experts Les programmes de diagnostic médical sont aujourd'hui capables de réaliser des diagnostics tout aussi fiables que les experts humains dans plusieurs spécialités médicales. On a même vu un expert

humain rejeter le diagnostic proposé par un système expert sur un cas difficile, avant de reconnaître son erreur après que le programme eût expliqué son jugement.

Logistique Un programme de plannification logistique fût utilisé par l'armée américaine pour coordonner ses véhicules, son équipement et ses soldats pendant la guerre du golfe en 1991. Grace à ce système, des problèmes qui dans le passé auraient pris des semaines pour être résolus le furent en quelques heures. On estime que l'économie réalisée par l'armée américaine à l'aide de ce programme fut supérieure au montant qu'elle avait investit dans la recherche en IA pendant 30 ans [3].

Robotique Aujourd'hui, des robots sont régulièrement utilisés pour réaliser plusieurs types d'interventions chirurgicales, aidant les médécins à effectuer des manipulations plus précises et ouvrant la possibilité de faire de la chirurgie à la distance.

On le voit sur ce dernier exemple, l'intelligence artificielle commence à avoir des applications très bénéfiques à notre société moderne. Mais comme peut en témoigner l'avant-dernier exemple, l'intelligence artificielle (tout comme la science en général), peut être aussi utilisée à des fins militaires, et il est donc important de surveiller de près les implications éthiques de cette jeune discipline.

3. Je ne sais pas si c'est une bonne ou une mauvaise chose ; je vous laisse juger vous-même...

Chapitre 2a: Introduction aux Algorithmes de Recherche pour la Résolution de Problèmes

Introduction et motivation

Comme le titre l'indique, le but de ce chapitre est d'introduire des algorithmes de recherche pour la résolution des problèmes. Nous commençerons par une définition formelle d'un problème (et d'une solution), et ensuite nous verrons un nombre d'exemples spécifiques pour illustrer cette définition et pour motiver la suite. Dans la deuxième partie du chapitre, nous détaillerons plusieurs algorithmes de recherche, d'abord les algorithmes de recherche dits non-informés, puis des algorithmes de recherche informés (ou algorithmes de recherche heuristiques), et enfin les algorithmes de recherche locale.

Définition formelle d'un problème

Dans le contexte de ce chapitre, un *problème* sera défini par les cinq éléments suivants :

1. un *état initial*
2. un ensemble d'*actions*
3. une *fonction de successeur*, qui définit l'état résultant de l'éxécution d'une action dans un état
4. un ensemble d'*états buts*
5. une *fonction de coût*, associant à chaque action un nombre non-négative (le coût de l'action)

Nous pouvons voir un problème comme un graphe orienté où les noeuds sont des états accessibles depuis l'état initial et où les arcs sont des actions. Nous appellerons ce graphe *l'espace des états*. Une *solution* sera un chemin de l'état initial à un état but. On dit qu'une solution est *optimale* si la somme des coûts des actions du chemin est minimale parmi toutes les solutions du problème.

Nous passons tout de suite à des exemples concrets qui devraient vous permettre de mieux comprendre cette définition.

Exemples de problèmes

Nous commençons par deux problèmes ludiques : le taquin et le problème des huit reines.

Exemple 1 (Le taquin)**.** Pour ceux que le connaissent pas, le taquin est une sorte de puzzle. Nous commençons avec une grille 3×3 de neuf cases où sont placées huit tuiles étiquetées par les nombres 1 à 8, une des cases restant vide. Une tuile située à côté de la case vide peut être déplacée vers cette case. L'objectif du jeu est d'arriver à obtenir une certaine configuration des tuiles dans la grille. Voici une formalisation de ce problème :

Etats Des états sont des configurations des huit tuiles dans les neuf cases de la grille. Voir FIG. 1 pour deux exemples.

Etat initial N'importe quel état pourrait être chosi comme l'état initial.

Actions Il y aura 4 actions possibles correspondant aux quatre façons de changer la position du carré vide : *haut*, *bas*, *gauche*, *droite* (remarquez que dans certaines configurations, il n'y aura que 2 ou 3 actions possibles).

Fonction de successeur Cette fonction spécifie les états résultants des différentes actions. Par exemple, la fonction va nous dire que l'exécution de l'action *droite* dans le premier état de FIG. 1 produira le deuxième état de FIG. 1.

Test de but L'état but est unique et fixé au début du jeu (n'importe quel état peut être choisi comme état but, même si en pratique il s'agit de remettre les nombres dans l'ordre).

Coût des actions Chaque déplacement d'une tuile a coût de 1 (pour trouver une solution avec le moins de déplacements).

Le taquin est souvent utilisé pour tester les algorithmes de recherche. En augmentant la taille de la grille, nous pouvons créer les problèmes de plus en plus complexes. Les algorithmes d'aujourd'hui arrivent à résoudre les taquins 3×3 et 4×4 (qui ont des espaces d'états de taille 181440 et d'environ $1,3$ milliard réspectivement), mais les instances du taquin 5×5 (avec un éspace d'états de taille 10^{25}) restent difficiles.

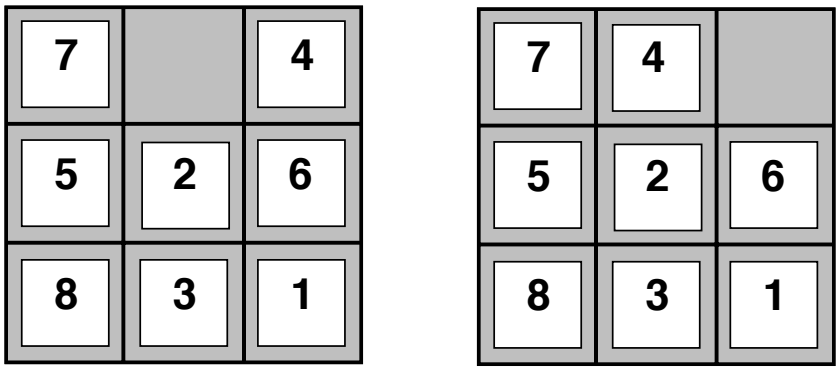

FIG. 1 – Deux états du jeu de taquin.

Exemple 2 (Huit reines)**.** L'objectif de ce jeu est de placer huit reines sur un échiquier (une grille 8×8)[1] tel qu' aucune reine attaque une autre reine, c'est à dire qu'il n'y a pas deux reines sur la même colonne, la même ligne, ou sur la même diagonale. La configuration donnée dans FIG. 2 n'est donc pas une solution parce qu'il y a deux reines sur la même diagonale. Voici une première formalisation :

Etats Toute configuration de 0 à 8 reines sur la grille.

Etat initial La grille vide.

Actions Ajouter une reine sur n'importe quelle case vide de la grille.

Fonction de successeur La configuration qui résulte de l'ajout d'une reine à une case spécifié à la configuration courante.

Test de but Une configuration de huit reines avec aucune reine sous attaque.

Coûts des actions Ce pourrait être 0, ou un coût constant pour chaque action - nous nous intéressons pas du chemin, seulement l'état but obtenu.

On peut réduire l'éspace d'états drastiquement (de 3×10^{14} à 2057 !) en observant qu'il est inutile de continuer de développer des configuations où il

[1]Ce problème, comme le taquin, peut être facilement généralisé : il suffit de remplacer 8 par n'importe quel nombre positif afin d'obtenir une suite de problèmes de complexité grandissante.

y a déjà à un conflit (comme on ne peut pas résoudre le conflit en ajoutant des reines supplémentaires). Voici les changements de notre formalisation :

Etat initials Configurations de n reines ($0 \leq n \leq 8$), une reine par colonne dans les n colonnes plus à gauche, avec aucun conflit.

Actions Ajouter une reine à une case vide dans la colonne vide la plus à gauche dans une façon à eviter un conflit.

Ces deux formalisations sont incrémentales car nous plaçons des reines une par une sur la grille. Mais ce n'est pas la seule façon de le faire. Une autre possibilité serait de commencer avec les huits reines sur la grille (une par colonne) et puis de changer la position d'une reine à chaque tour.

Etat initials Une configuration de 8 reines telle qu'il n'y a qu'une seule reine par colonne

Etat initial Un état choisi aléatoirement.

Actions Changer la position d'une reine dans sa colonne.

Notons qu'avec les characterisations incrémentales nous ne tombons jamais sur un état déjà visité, mais ce n'est pas le cas avec cette dernière formalisation (car nous pouvons changer la position d'une reine et après la remettre dans sa position d'origine) ni avec le taquin (où nous pouvons très bien déplacer une tuile et la remettre à sa place au coup suivant). Dans ces derniers cas, il nous faut faire attention pour ne pas explorer une deuxième fois un chemin déjà exploré ou de tourner en boucle.

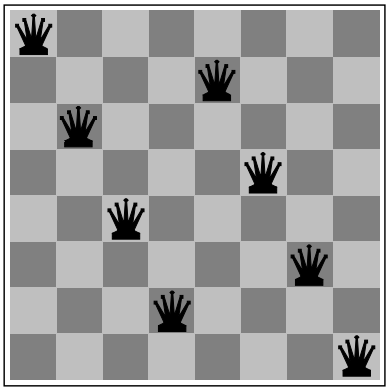

FIG. 2 – Une presque solution pour le problèmes des huit reines.

Nous remarquons que ces deux problèmes sont de nature assez différentes. Avec le taquin, nous savons depuis le début quel état nous voulons, et la difficulté est de trouver une séquence d'actions pour l'atteindre. En ce qui concerne le problème des huit reines, nous ne sommes pas intéréssés par le chemin mais seulement par l'état but obtenu. Ces deux jeux sont des exemples de deux grandes classes de problèmes étudiés en IA : des problèmes de planification et des problèmes de satisfaction de contraintes. Nous examinons plus en détail ces deux classes de problèmes.

Exemple 3 (Problèmes de planification)**.** Dans un problème de planification, nous cherchons une suite d'actions pour relier l'état initial à un état but. Voici une formalisation standard du problème[2] :

Etats Un ensemble d'états du monde qui correspondent aux différentes valuations d'un ensemble de propositions. Par exemple, considérons un cas très simple avec une seule proposition **allumée** (qui dit que la lumière est allumée). Nous avons deux états du monde possibles : **allumée**=*vrai* et **allumée**=*faux*.

Etat initial L'état actuel du monde, e.g.[3] **allumée**=*vrai*.

Actions Un ensemble d'actions autorisées, e.g. *actionner l'interrupteur*.

Fonction de successeur Cette fonction va nous dire quelles actions sont possibles dans chaque état du monde et comment les actions changent l'état du monde, e.g. l'action *actionner l'interrupteur* va faire passer de l'état **allumée**=*vrai* à l'état **allumée**=*faux* et inversement.

Test de but Théoriquement, ce pourrait être n'importe quel ensemble d'états, mais souvent nous définissons le but comme l'ensemble des états satisfaisants une certaine valuation partielle des propositions.

Coût des actions Pour essayer de trouver des plans les plus courts, nous pouvons mettre tous les coûts à 1. Nous pouvons aussi définir les coûts des actions en fonction de leur durée, de leur coût en argent, de la quantité de ressources qu'elles consomment, etc.

Les problèmes de planification sont courants dans la vie de tous les jours. Par exemple, considérons le problème de trouver le chemin le plus court pour atteindre une destination. C'est un problème tellement courant qu'il y a maintenant des GPS dédiés à cette tâche. Citons également les systèmes

[2]Cette formalisation repose sous de nombreux hypothèses. Par exemple, nous supposons que les actions sont déterministes et que l'agent connaît son état d'origine, deux hypothèses qui ne seront pas toujours vérifées dans les problèmes de ce type. Nous discutérons un peu plus de ces complexités dans le chapitre dédié à la planification.

[3]abréviation de la locution latine *exempla gratia*, qui signifie "par exemple" ; je l'utiliserai fréquemment dans ce cours

de planification de voyage qui cherchent des itinéraires en transports en commun[4], en train ou en avion. Formalisons ce dernier exemple :

Etas Chaque état est composé d'un aéroport et la date et l'heure actuelle.

Etat initial L'aéroport d'où part le client, la date de départ souhaitée, et l'heure à partir de laquelle le client peut partir.

Actions Ce sont des vols d'un aéroport à un autre.

Fonction de successeur Les actions possibles sont des vols qui partent de l'aéroport de l'état actuel, plus tard que l'heure actuelle (en fait, nous devrons aussi exiger une certaine durée entre l'arrivée dans un aéroport et le prochain vol, sinon le client risque de rater sa correspondance!!).

Test de but Les états où le client est à l'aéroport de sa destination.

Coût des actions Cela va dépendre des préférences du client. Ce pourrait être 1 pour chaque action (pour minimiser la nombre de connections), ou la durée des vols (pour minimiser la durée du voyage), ou le prix des trajets (pour trouver le voyage le moins cher).

Un problème très similaire est celui du routage dans les réseaux, où il faut trouver les chemins pour envoyer des paquets. Enfin, les planificateurs sont aussi utilisés dans l'industrie pour trouver des manipulations physiques pertmettant à un robot de construire des objets complexes à partir de composants donnés[5].

Exemple 4 (Satisfaction de contraintes)**.** Un problème de satisfaction de contraintes (CSP) consiste en un ensemble de variables, des ensembles de valeurs permises pour chaque variable, et un ensemble de contraintes (sur les combinaisons des valeurs des variables). L'objectif est de trouver une valuation telle que chaque contrainte est satisfaite. Voici une formalisation incrémentale d'un CSP :

Etat initials Des valuations partielles (c'est à dire, des choix de valeurs pour un sous-ensemble des variables)

Etat initial La valuation vide (où nous n'avons pas encore choisi des valeurs)

Actions Choisissez une valeur pour une des variables restantes.

Fonction de successeur Nous ajoutons la nouvelle valeur à la valuation actuelle.

[4]essayez par exemple celui de `www.rtm.fr`

[5]Ceux qui ont déjà essayé de constuire un meuble Ikea apprécieront la difficulté de cette tâche!!

Test de but Un état but est une valuation complète telle que chaque contrainte est satisfaite.

Coût des actions Cela peut dépendre du problème en question, mais en général nous devrons donner le même coût pour chaque action comme les solutions sont toutes aussi bonnes.

Nous pouvons également donner une formalisation à partir des états complets :

Etats Une valuation des variables.

Etat initial Une valuation quelconque.

Actions Changez la valeur d'une variable.

Gèrer des pistes d'atterissage à l'aéroport, trouver un plan de table pour un mariage, créer un emploi du temps à l'université.... ce sont tous des problèmes de satisfaction de contraintes. Pour ce dernier exemple, les variables sont des cours, des valeurs possibles sont des triplets (jour, heure, salle), et des contraintes de toute sorte (il faut qu'un cours INF100 soit enseigné le vendredi, il faut pas mettre INF101 et INF102 en même temps, etc.). Nous donnons une formalisation incrémentale de cet exemple :

Etats Un emploi de temps partiel (c'est à dire, un choix d'une triplet (jour, heure, salle) pour un sous-ensemble des cours).

Etat initial Un emploi de temps vide.

Actions Chossisez une triplet (jour, heure, salle) pour un cours qui n'est pas encore sur la grille.

Fonction de successeur L'emploi de temps résultant.

Test de but Un emploi de temps contenant tous les cours et satisfaisant toutes les contraintes.

Coût des actions Un coût constant.

Dans tous ces problèmes considérés jusqu'à présent, il n'y avait qu'un seul agent[6] qui pouvait choisir librement ses actions. Mais souvent nos décisions sont conditionnées par les actions des autres. C'est notamment le cas pour les jeux à plusieurs joueurs, que nous considérons maintenant.

Exemple 5 (Jeux)**.** Les jeux à plusieurs joueurs sont plus compliqués que les problèmes que nous avons vu parce que l'agent ne peut pas choisir les actions des autres joueurs. Donc, au lieu de chercher un chemin qui relie l'état initial à un état but (qui contiendrait des actions des adversaires, dont

[6]un *agent* est un terme important en IA qui désigne une entité capable d'interagir avec son environnement.

nous n'avons pas le contrôle), nous allons chercher une *stratégie*, c'est à dire un choix d'action pour chaque état où pourrait se trouver l'agent. Voici une formalisation d'un jeu à plusieurs joueurs :

Etats Des configurations du jeu plus le nom de joueur dont c'est le tour.

Etat initial La configuration initiale plus le nom du joueur qui commence.

Actions Les coups légaux pour le joueur dont c'est le tour.

Fonction de successeur La nouvelle configuration du jeu, et le nom de joueur dont c'est le tour.

Test de but Les configurations gagnantes pour le joueur.

Coût des actions Cela dépendra du jeu en question.

Pour le jeu d'échecs, nous aurions la formalisation suivante :

Etats Des états sont composés d'une configuration de l'échiquier plus le nom du joueur dont c'est le tour[7].

Etat initial La configuration standard pour le début d'une partie d'échecs.

Actions Ce sont des coups qui peuvent être joués par le joueur dont c'est le tour dans la configuration actuelle du jeu (y compris l'action d'abandonner le jeu).

Fonction de successeur La configuration du jeu évolue selon le coup choisi, et c'est maintenant à l'autre joueur de jouer.

Test de but Il y a beaucoup de façons de terminer une partie, la plus connue étant l'échec et mat (je ne vais pas les détailler ici, ceux que cela intéresse pourront se documenter sur le sujet).

Coût des actions Nous pouvons les mettre à 1 pour chercher les coups qui ammènent le plus vite à une configuration gagnante.

Structure générale d'un algorithme de recherche

La plupart des algorithmes de recherche suivent à peu près le même schéma : nous commençons toujours dans l'état initial et puis nous exécutons les étapes suivantes en boucle jusqu'à terminaison :

- s'il n'y a plus d'états à traiter, renvoyez **echec**
- sinon, choisir un des états à traiter ($\star$)
  - si l'état est un état but, renvoyez la solution correspondante

[7]Ce n'est pas complètement exact, mais seuls les vrais joueurs d'échecs comprendront pourquoi (il faudrait en théorie ajouter l'historique des coups qui ont été joués pour arriver à l'état actuel)

– sinon, supprimer cet état de l'ensemble des états à traiter, et le remplacer par ses états successeurs

Ce qui va différencier les différents algorithmes est la manière dont on effectue le choix à l'étape ($\star$).

Evalution des algorithmes de recherche

Dans la suite, nous allons différents d'algorithmes de recherche. Comment pouvons-nous les comparer ? Voici quatre critères que nous allons utiliser pour comparer les différents algorithmes de recherche :

Complexité en temps Combien du temps prend l'algorithme pour trouver la solution ?

Complexité en espace Combien de mémoire est utilisée lors de la recherche d'une solution ?

Complétude Est-ce que l'algorithme trouve toujours une solution s'il y en a une ?

Optimalité Est-ce que l'algorithme renvoie toujours des solutions optimales ?

Chapitre 2b: Algorithmes de Recherche

Un algorithme de recherche générique

Avant de considérer des algorithmes spécifiques, nous présentons un algorithme de recherche générique, dont le pseudo-code est donné dans Figure 1.

```
fonction RECHERCHE(état_initial, SUCCESSEURS, TEST_BUT, COÛT)
    noeuds_à_traiter ← CRÉER_LISTE(CRÉER_NOEUD(état_initial, [], 0))
    boucle
        si VIDE ?(noeuds_à_traiter) alors renvoyer échec
        noeud ← ENLEVER_PREMIER_NOEUD(noeuds_à_traiter)
        si TEST_BUT(ÉTAT(noeud)) = vrai
            alors renvoyer CHEMIN(noeud), ÉTAT(noeud)
        pour tout (action, état) dans SUCCESSEURS(ÉTAT(noeud))
            chemin ← [action, CHEMIN(noeud)]
            coût_du_chemin ← COÛT_DU_CHEMIN(noeud) + COÛT( état)
            s ← créer_noeud(état, chemin, coût_du_chemin)
            INSÉRER(s, noeuds_à_traiter)
```

FIGURE 1 – Un algorithme de recherche générique.

L'algorithme prend en entrée la description d'un problème : un état initial, une fonction de successeurs, un test de but, et une fonction de coût. La première étape de l'algorithme est d'initialiser une liste [1] de noeuds à traiter, un noeud étant composé d'un état, d'un chemin, et du coût du chemin (voir Figure 2). Nous commençons avec un seul noeud correspondant à l'état intial.

A chaque itération de la boucle, nous vérifions si la liste de noeuds à traiter est vide. Si c'est le cas, nous avons examiné tous les chemins possibles sans pour autant trouver une solution, donc l'algorithme renvoie "échec". Si la liste contient encore des noeuds, nous sortons le premier noeud de la liste.

1. Pour nous, une liste sera une suite ordonnée d'éléments, dans laquelle on peut insérer de nouveaux éléments (la façon dont les éléments seront insérés dépendra de l'algorithme de recherche en question) et de supprimer des éléments en début de suite.

Si l'état de ce noeud est un état but, c'est gagné, et nous renvoyons l'état et le chemin qui permet d'accéder à ce noeud but. Dans le cas contraire, la recherche se poursuit : nous produisons les successeurs du noeud et les insérons dans la liste de noeuds à traiter.

Recherche non-informée

Dans cette section, nous introduisons des algorithmes de recherche *non-informés*. Ces algorithmes sont ainsi nommés parce qu'ils ne disposent pas d'informations supplémentaires pour pouvoir distinguer des états prometteurs (ce n'est pas le cas par exemple des programmes joueurs d'échecs qui ne peuvent explorer toutes les possibilités, et se concentrent donc à chaque étape sur un petit nombre de coups qui leur semblent être les meilleurs). En l'absence de telles informations, ces algorithmes font une recherche exhaustive de tous les chemins possibles depuis l'état initial.

Parcours en largeur

Le parcours en largeur est un algorithme de recherche très simple : nous examinons d'abord l'état initial, puis ses successeurs, puis les successeurs des successeurs, etc. Tous les noeuds d'une certaine profondeur sont examinés avant les noeuds de profondeur supérieure. Pour implémenter cet algorithme, il suffit de placer les nouveaux noeuds systématiquement à la fin de la liste de noeuds à traiter. Le fonctionnement du parcours en largeur sur un exemple

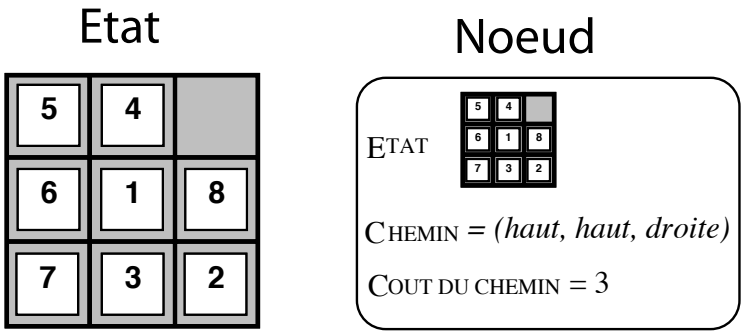

FIGURE 2 – Un noeud est composé d'un état, un chemin (une suite d'actions qui relie l'état à l'état initial), et le coût du chemin.

est présenté sur la Figure 3.

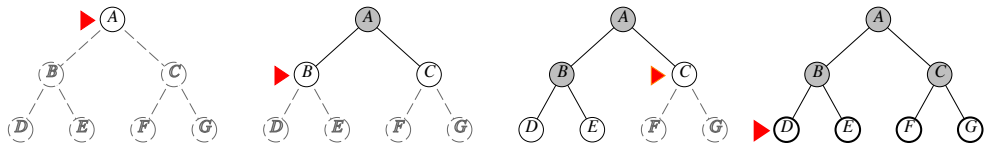

FIGURE 3 – Le parcours en largeur pour un arbre binaire simple. Nous commençons par l'état initial A, puis nous examinons les noeuds B et C de profondeur 1, puis les noeuds D, E, F, et G de profondeur 2.

Le parcours en largeur est un algorithme de recherche complet [2] à condition que le nombre de successeurs des états soit toujours fini (ce qui est très souvent le cas dans les problèmes courants). Pour voir pourquoi, soit p le nombre minimal d'actions nécessaires pour atteindre un état but depuis l'état initial. Comme nous examinons les noeuds profondeur par profondeur, et qu'il n'y a qu'un nombre fini de noeuds à chaque profondeur (ce ne serait pas le cas si un noeud pouvait avoir un infini de successeurs), nous atteindrons forcément le niveau p.

Le parcours en largeur trouve toujours un état but de plus petite profondeur possible. Donc si nous cherchons une solution quelconque, ou une solution avec le moins d'actions possibles, le parcours en largeur est optimal. Un peu plus spécifiquement, le parcours en largeur est optimal quand le coût du chemin ne dépend que du nombre d'actions de ce chemin. Par contre le parcours en largeur n'est pas optimal dans le cas général (Exercice).

Considérons ensuite la complexité de cet algorithme. Supposons que chaque état possède s successeurs et que p soit le nombre minimal d'actions pour relier l'état initial à un état but. Dans le pire des cas, nous allons examiner tous les noeuds de profondeur au plus p afin de trouver l'état but qui se trouve à cette profondeur. Nous allons alors produire

$$1 + s + s^2 + s^3 + ... + s^p + (s^{p+1} - s)$$

noeuds. La quantité $s^{p+1} - s$ dans cette formule correspond au nombre de noeuds de profondeur $p+1$ qui ont été produits lors de l'examen des noeuds

2. Rapellons que cela signifie que nous trouvons une solution s'il y en a une.

Profondeur	Noeuds Générés	Temps		Mémoire	
2	1100	0,11	sec	1	mega-octet
4	111 100	11	sec	106	mega-octet
6	10^7	19	min	10	giga-octet
8	10^9	31	h	1	tera-octet
10	10^{11}	129	jours	101	tera-octets
12	10^{13}	35	ans	10	peta-octets

FIGURE 4 – Voici le nombre de noeuds générés et le temps et la mémoire utilisés par le parcours en largeur pour un nombre de successeurs fixé à 10 (supposant 10,000 noeuds par seconde et 1000 octets par noeud).

de profondeur p moins les successeurs du dernier noeud (comme le dernier noeud est le bon, nous ne produisons pas ses successeurs). Le parcours en largeur a donc une complexité en temps de $\mathcal{O}(s^{p+1})$. La complexité en espace est aussi en $\mathcal{O}(s^{p+1})$ puisque nous avons au plus $1 + s^{p+1} - s$ noeuds en mémoire lors de l'examen du dernier noeud.

La Figure 4 montre le temps et la mémoire nécessaires au parcours en largeur pour un problème ayant 10 successeurs par état ($s = 10$). Nous voyons que même pour $p = 6$, le parcours en largeur utilise 10 giga-octets de mémoire, et pour $p = 8$, il lui faut 1 tera-octet ! Le temps de calcul pose aussi problème : l'algorithme prend 35 ans pour trouver une solution de profondeur $p = 12$. Cette haute complexité en temps et en espace restreint l'utilisation du parcours en largeur aux petits problèmes.

Parcours en profondeur

Le parcours en profondeur suit le chemin courant le plus longtemps possible. Il est facile à implémenter : il faut tout simplement mettre des successeurs du noeud courant au début de la liste de noeuds à traiter. La Figure 5 montre le fonctionnement de cet algorithme sur un petit exemple.

Le parcours en profondeur n'est pas complet parce que l'algorithme peut continuer sur un chemin infini, ignorant complètement un état but qui se trouve sur un autre chemin. Si par contre, nous n'avons qu'un nombre fini de chemins possibles (ce qui n'est pas souvent le cas), le parcours en profondeur sera complet. L'algorithme n'est pas optimal : il n'y a rien qui garantie que le premier état but trouvé sera le bon. Considérons maintenant la complexité. Si nous avons s successeurs et une profondeur maximale de m actions, dans le pire des cas nous aurions besoin d'examiner tous les s^m noeuds pour trouver une solution. Sur ces trois premiers critères, il est évident que le parcours en

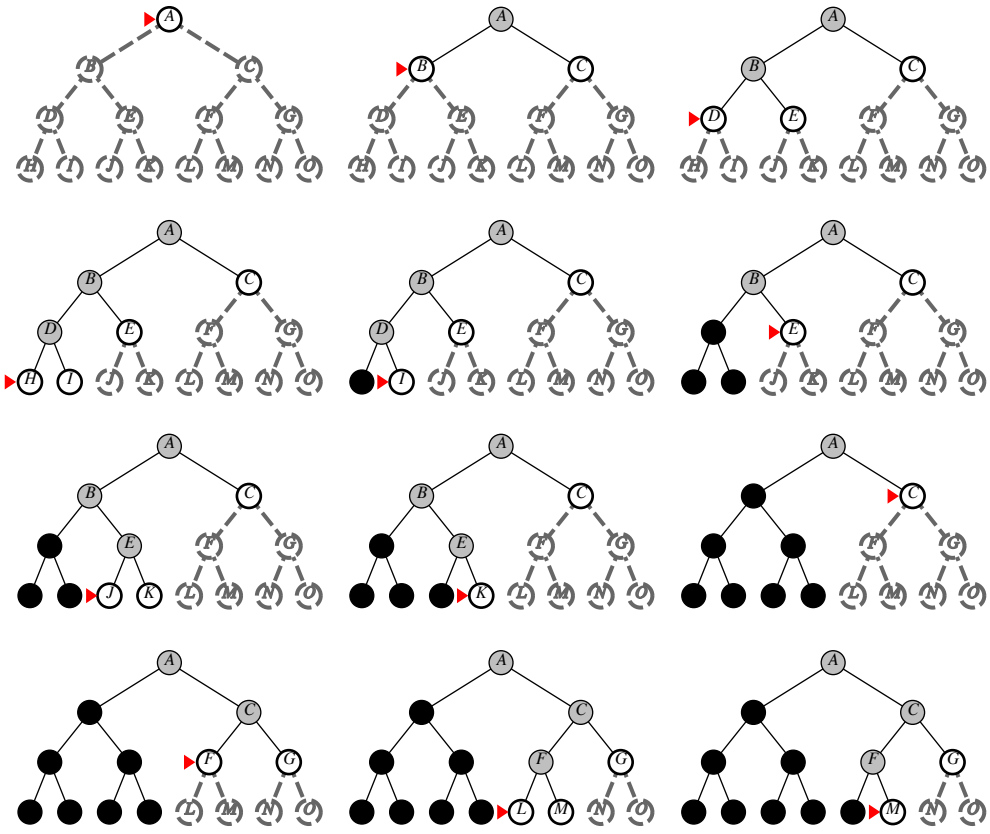

FIGURE 5 – Le parcour en profondeur pour un arbre binaire simple. En gris le chemin en cours d'examen, en blanc les noeuds non-explorés, et en noir les noeuds explorés et rejetés.

largeur sort gagnant. Alors pourquoi s'intérésser au parcours en profondeur ? Le principal avantage du parcours en profondeur reste en sa faible complexité en espace. Pour un problème ayant s successeurs et un profondeur maximal de m actions, il nous faut garder au plus $1 + (m * (s - 1))$ noeuds en mémoire (correspondant au chemin actuellement en développement plus les successeurs de profondeur 1 que nous n'avons pas encore traités, les successeurs de profondeur 2 que nous n'avons pas encore traités, etc.). La complexité en espace est donc de $\mathcal{O}(s * m)$. Même quand s et m sont grands, la mémoire nécéssitée par le parcours en profondeur reste raisonnable, ce qui nous permet de traiter des problèmes qui ne sont pas abordables par le parcours en largeur.

Parcours en profondeur limitée

Nous venons de voir que le parcours en profondeur n'est pas très bien adapté aux problèmes où la longeur des chemins n'est pas bornée parce que nous risquons de suivre aveuglément un chemin infini qui ne mène pas à un état but. Une façon naïve d'éviter ce problème serait de fixer une profondeur maximale et de ne pas considérer les chemins de profondeur supérieur à cette limite. L'algorithme résultant, nommé parcours en profondeur limitée, va toujours terminer (à condition bien sûr que le nombre de successeurs soit fini) ce qui n'était pas le cas pour le parcours en profondeur simple.

Le parcours en profondeur limitée est complet si la profondeur maximale est supérieure à la profondeur minimale des solutions. Mais comme nous savons pas en général quelle profondeur sera suffisante, nous ne saurons pas si l'algorithme est complet ou non pour une profondeur donnée. Comme c'était le cas pour le parcours en profondeur classique, le parcours en profondeur limitée n'est pas optimal en général. Pour une profondeur maximale fixée à m et s successeurs par état, la complexité en temps sera s^m (dans les pire des cas il faut examiner chacun des s^m chemins de profondeur au plus m). La complexité en espace est d'ordre $\mathcal{O}(s * m)$.

Parcours en profondeur itérée

Dans la dernière section, nous avons introduit le parcours en profondeur limitée dont l'inconvénient principal est la difficulté de bien choisir la borne de profondeur. Le parcours en profondeur itérée permet de remédier (de façon un peu brutale !) à cet inconvénient : comme nous ne savons pas quelle borne de profondeur choisir, nous allons les essayer les unes après les autres. Nous effectuons donc un parcours en profondeur limitée avec une borne de 1, puis un parcours de profondeur avec une borne de 2, et nous continuons

à augmenter la borne jusqu'à ce que l'on trouve une solution. La Figure 6 montre le déroulement de l'algorithme sur un exemple.

Comme le parcours en largeur, le parcours en profondeur itérée est complet — à condition que le nombre de successeurs soit toujours fini — et optimal quand le coût d'un chemin ne dépend que du nombre d'actions. Pour un problème ayant s successeurs par état et une solution de profondeur p, nous allons examiner $p+1$ fois le noeud unique de profondeur 0, p fois les noeuds de profondeur 1, $p-1$ fois les noeuds de profondeur 2, ..., et une seule fois les noeuds de profondeur p. Le nombre de noeuds génére lors de la recherche est donc

$$(p+1) + (p)s + (p-1)s^2 + (p-2)s^3 + ... + (1)s^p$$

ce qui donne une complexité en temps en $\mathcal{O}(s^p)$. Remarquons que même s'il semble inefficace de traiter les mêmes noeuds plusieurs fois, la complexité en temps du parcours en profondeur itérée est en fait moins importante que celle du parcours en largeur (où la complexité est $\mathcal{O}(s^{p+1})$ parce que nous produisons des noeuds de profondeur $p+1$ lors de l'examen des noeuds de profondeur p) et que celle du parcours en profondeur classique (où nous pouvons examiner les noeuds de profondeur supérieure à p). Quant à la complexité en espace, nous gardons au plus $1 + (p * (s-1))$ noeuds en mémoire lors de l'examen des noeuds de profondeur p, donc la complexité en espace est en $\mathcal{O}(s * p)$.

Le parcours en profondeur itérée combine donc les avantages du parcours en largeur (complétude et optimalité) et du parcours en profondeur (faible complexité en espace). C'est pour cette raison que le parcours en profondeur itérée est considéré aujourd'hui comme le meilleur algorithme de recherche non-informée pour les problèmes de grande taille où la profondeur des solutions est inconnue.

Etats redondants

Les algorithmes que nous venons de présenter peuvent visiter plusieurs fois le même état pendant la même recherche. Il y a deux types de redondances possibles : d'une part, il y a des états qui peuvent être visités deux fois sur le même chemin (par exemple, au taquin, si vous jouez droite puis gauche, vous revenez à l'état initial), et d'autre part, il peut y avoir plusieurs chemins différents qui amènent au même état [3]. Dans cette section, nous considérons les façons d'éviter de telles redondances.

3. Les plus attentifs auront remarqué qu'il ne s'agit pas vraiment d'un type différent mais plutôt d'un cas plus général que le premier type.

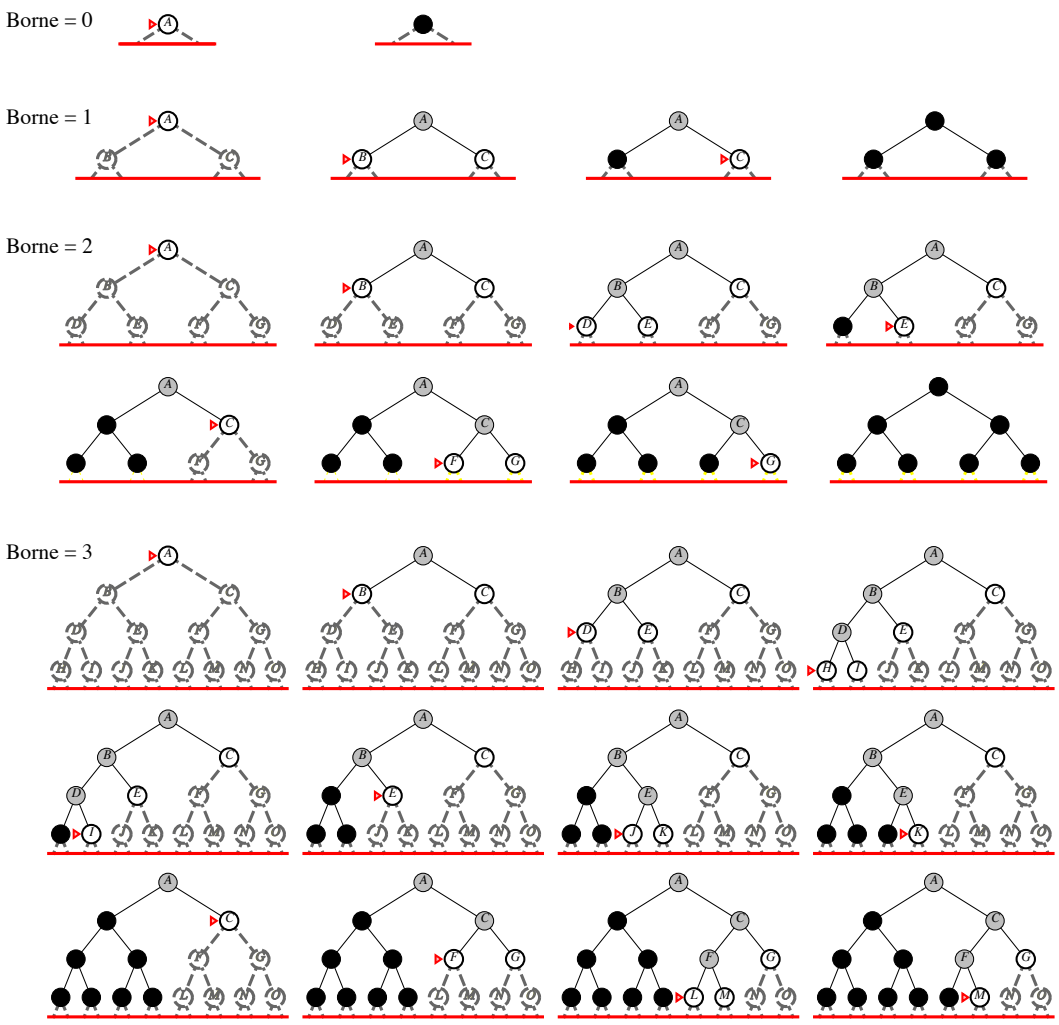

FIGURE 6 – Le parcours en profondeur itérée pour un arbre binaire simple.

Il est relativement simple d'éviter le première type de redondance. Il suffit d'examiner au fur et à mesure les chemins des noeuds à insérer et de ne jamais ajouter à la liste de noeuds à traiter des noeuds dont les chemins contiennent plusieur fois le même état. Il est possible de modifier les différents algorithmes de telle façon que cette modification ne change pas significativement la complexité en espace.

Par contre, si nous voulons éviter le deuxième type de redondance, cela signifie qu'il faut se rappeler de tous les états déjà visités et de ne garder qu'un seul chemin par état. Plus spécifiquement, il faut gérer une liste des états déjà visités et y ajouter les états quand ils sont visités pour la première fois. Avant de générer les successeurs d'un noeud, il faut vérifier si l'état successeur est dans la liste des états déjà visités (et si c'est le cas, nous ne générons pas de successeurs). Cette modification ne changera pas trop la complexité en espace pour le parcours en largeur (qui souffre déjà d'une haute complexité en espace). Par contre, pour le parcours en profondeur et le parcours en profondeur itérée qui gardent normalement peu de noeuds en mémoire cette modification va augmenter très significativement la complexité en espace.

Recherche Heuristique

Les algorithmes que nous avons vus dans la dernière section font une recherche exhaustive de tous les chemins possibles, ce qui les rend inefficaces voire inutilisables sur les problèmes de grande taille. Dans cette section, nous présentons les algorithmes de recherche heuristiques qui utilisent des informations supplémentaires pour pouvoir mieux guider la recherche.

Tout algorithme de recherche heuristique dispose d'une fonction d'évaluation f qui détermine l'ordre dans lequel les noeuds sont traités : la liste de noeuds à traiter est organisée en fonction des f-valeurs des noeuds, avec les noeuds de plus petite valeur en tête de liste.

A priori, il n'y a pas vraiment de restriction sur la nature de la fonction d'évalution, mais souvent elle a comme composante une fonction heuristique h où

$$h(n) = \text{coût estimé du chemin de moindre coût reliant } n \text{ à un état but}$$

Notons que la fonction heuristique prend un noeud en entrée mais sa valeur ne dépend que de l'état associé au noeud. Et bien sûr, $h(n) = 0$ si n est un état but.

Prenons par exemple le problème suivant : nous sommes à Arad en Roumanie et il nous faut atteindre (en parcourant la plus petite distance possible)

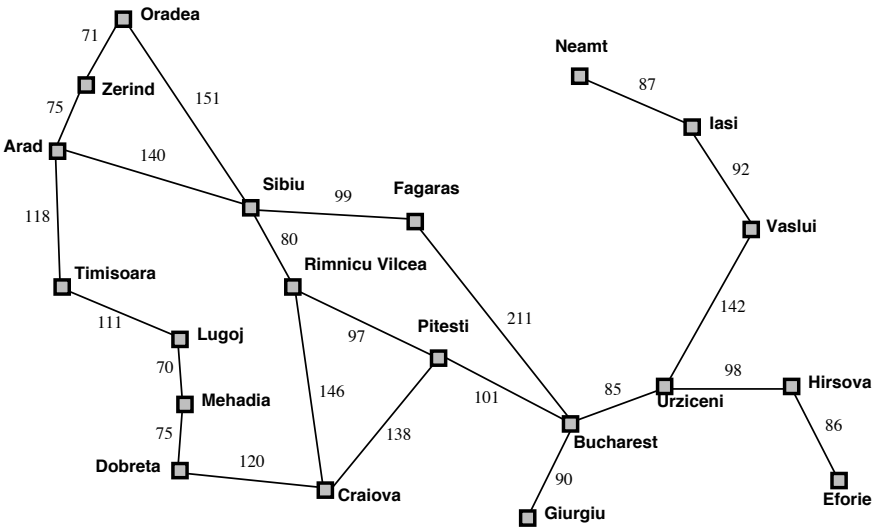

FIGURE 7 – Une carte de route simplifié de la Roumanie.

Bucarest. La Figure 7 présente les principales routes de la Roumanie et les distances associées. Soit $d(A, B)$ la distance routière entre la ville A et la ville B, c'est-à-dire le plus court chemin entre la ville A et la ville B. On a par exemple $d(Zerind, Sibiu) = 215$. Pour mesurer à quel point nous sommes proches du but lorsque nous sommes dans la ville X, il serait bon de connaître $d(X, Bucarest)$, et on aimerait alors prendre $h(X) = d(X, Bucarest)$. Le problème est que l'on n'a pas accès à la fonction d, et que son calcul n'est pas trivial (l'exemple ici étant de petite taille, calculer d serait faisable, mais sur un graphe de grande taille cela est beaucoup plus difficile). C'est ici qu'intervient l'heuristique, c'est-à-dire l'approximation : on approxime $d(X, Bucarest)$ par $h(X)$=distance à vol d'oiseau entre la ville X et Bucarest (qui elle est très facilement calculable à l'aide d'une carte routière et d'un décimètre). Les valeurs de cette heuristique sont fournies dans la Figure 8. Comme les distances par la route sont toujours supérieures à la distance à vol d'oiseau, $h(X)$ n'est qu'une approximation du coût réel. Si notre heuristique était parfaite, nous n'aurions même pas besoin de faire une recherche ! (Exercice)

Arad	366	Mehadia	241
Bucarest	0	Neamt	234
Craiova	160	Oradea	380
Dobreta	242	Pitesi	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

FIGURE 8 – Des valeurs heuristiques : distances à vol d'oiseau de Bucarest.

Best-first search (meilleur d'abord)

L'idée de l'algorithme "best-first search" (à ma connaissance, il n'y a pas de terme français couramment utilisé) est d'examiner les noeuds qui semblent les plus proches d'un état but, dans l'espoir d'aboutir plus vite à une solution. Un peu plus formellement, la stratégie employée par best-first search consiste à utiliser la fonction heuristique h comme fonction d'évaluation (c'est-à-dire qu'on prend $f(n) = h(n)$).

Figure 9 montre le déroulement du best-first search sur le problème de trouver un chemin de Arad à Bucarest que nous venons de présenter. Nous commençons dans l'état initial Arad, puis nous considérons les trois successeurs possibles : Sibiu (avec valeur 253), Timisoara (avec valeur 329), et Zerind (avec valeur 374). Le noeud Sibiu a la plus petite valeur heuristique, c'est donc lui qui sera examiné ensuite. Comme Sibiu n'est pas un état but, nous ajoutons ses quatre successeurs à la liste de noeuds à traiter : Arad (366), Fagaras (176), Oradea (380), et Rimnicu Vilcea (193). Le noeud le plus prometteur (parmi tous les noeuds restants à traiter) est Fagaras. Comme Fagaras n'est pas un état but, nous ajoutons ses deux successeurs Sibiu (253) et Bucarest (0) à la liste de noeuds à traiter. Nous choissisons alors Bucarest puisque ce noeud possède la plus petite valeur heuristique. Comme ce noeud correspond à un état but, nous arrêtons la recherche et renvoyons le chemin trouvé (Arad, Sibiu, Fagaras, Bucarest).

Best-first search n'est ni complet ni optimal. Il n'est pas complet car l'algorithme peut tourner en boucle mêmes'il existe une solution. Pour voir qu'il n'est pas optimal, il suffit de regarder l'exemple que nous venons d'examiner : le chemin (Arad, Sibiu, Fagaras, Bucarest) retourné par l'algorithme est plus long que le chemin (Arad, Rimnicu Vilcea, Pitesti, Bucarest). Pour

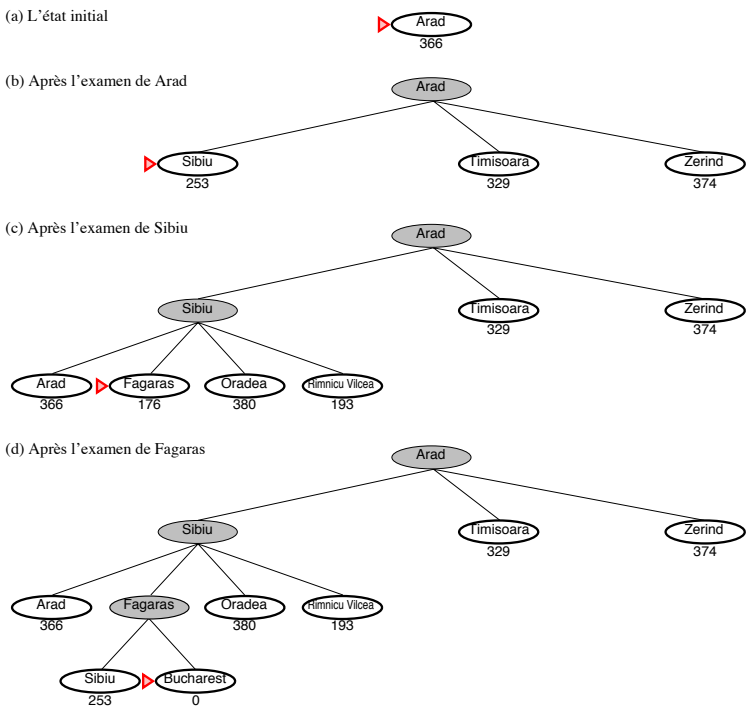

FIGURE 9 – Best-first search pour trouver un chemin de Arad à Bucarest.

une profondeur maximale de p et s successeurs, la complexité en temps et en espace de best-first search sont toutes les deux en $\mathcal{O}(s^p)$ dans les pire des cas, même si la complexité dépend en pratique de la qualité de la fonction heuristique.

Recherche A*

Best-first search donne la préférence aux noeuds dont les états semblent les plus proche d'un état but, mais il ne prend pas en compte les coûts des chemins reliant l'état initial à ces noeuds. Néanmoins, c'est une information très pertinente car le coût d'un chemin passant par un noeud n est la somme du coût de chemin entre l'état initial et n et le coût du chemin reliant n à un état but. C'est cette idée qui est à la base de la recherche A* . Si nous appelons $g(n)$ le coût du chemin entre l'état initial et n, la fonction d'évaluation utilisée par la recherche A* est donnée par la formule suivante :

$$f(n) = g(n) + h(n)$$

Comme $g(n)$ est le coût réel associé au chemin entre l'état initial et n et que $h(n)$ est une estimation du coût du chemin entre n et un état but, la fonction d'évaluation f donne une estimation du coût de la meilleure solution passant par le noeud n.

Figure 10 montre quelques étapes dans le déroulement de la recherche A* sur notre problème de trouver un chemin de Arad à Bucarest. Nous commençons dans l'état initial Arad. Comme Arad n'est pas un état but, nous considérons ses trois successeurs et comparons leurs valeurs de f : Sibiu (393= 140 + 253), Timisoara (447 = 118 + 329), et Zerind (449 = 75 + 374). Nous choissisons Sibiu comme ce noeud a la plus petite valeur d'évaluation parmi tous les noeuds restants à traiter. Nous ajoutons les quatre successeurs de Sibiu à la liste de noeud à traiter : Arad (646= 280 + 366), Fagaras (415= 239 + 176), Oradea (671 = 291 + 380), et Rimnicu Vilcea (413= 220 + 193). Le noeud le plus prometteur est Rimnicu Vilcea, donc nous ajoutons ses trois successeurs : Craiova (526= 366 + 163), Pitesti (417= 317 + 100), et Sibiu (553= 300 + 253) [4]. Nous comparons les différents noeuds à traiter, et nous choissisons Fagaras comme il a la plus petite valeur d'évaluation. Nous ajoutons les deux successeurs de Faragas à la liste de noeuds à traiter : Sibiu (591= 338 + 253) et Bucarest (450= 450 + 0). Remarquons que Bucarest est dans la liste de noeuds à traiter, mais nous n'arrêtons pas encore parce que nous avons toujours un espoir de trouver une solution de moindre coût. Nous continuons alors avec Pitesti qui a une valeur d'évaluation de 417 (qui est inférieur au coût de chemin à Bucarest que nous venons de trouver). Les trois successeurs de Pitesti sont ajoutés à la liste de noeuds à examiner : Bucarest (418=418 + 0), Craiova (615= 455 + 160), et Rimnicu Vilcea (607= 414 + 193). Maintenant c'est le nouveau noeud Bucarest qui a la meilleure valeur d'évaluation, et comme Bucarest est un état but, l'algorithme termine et renvoie le chemin (Arad, Rimnicu Vilcea, Pitesti, Bucarest) associé à ce noeud.

L'algorithme de recherche A* est complet et optimal s'il y a un nombre fini de successeurs (on commence à avoir l'habitude....) et si nous plaçons une certaine restriction sur la fonction heuristique h. Il faut que la fonction h soit *admissible*, c'est à dire que la valeur $h(n)$ ne doit jamais être supérieure au coût réel du meilleur chemin entre n et un état but. Notre heuristique prenant les distances à vol d'oiseau est un exemple d'une fonction heuristique admissible parce que la distance à vol d'oiseau (la valeur heuristique) n'est jamais supérieure à la distance par la route (le vrai coût).

Si nous voulons utiliser la technique proposée pour éliminer les noeuds

4. Notons que le noeud Sibiu que nous venons d'ajouter n'est pas le même que nous avons traité à tout à l'heure parce que ces deux noeuds ont des chemins et coûts différents.

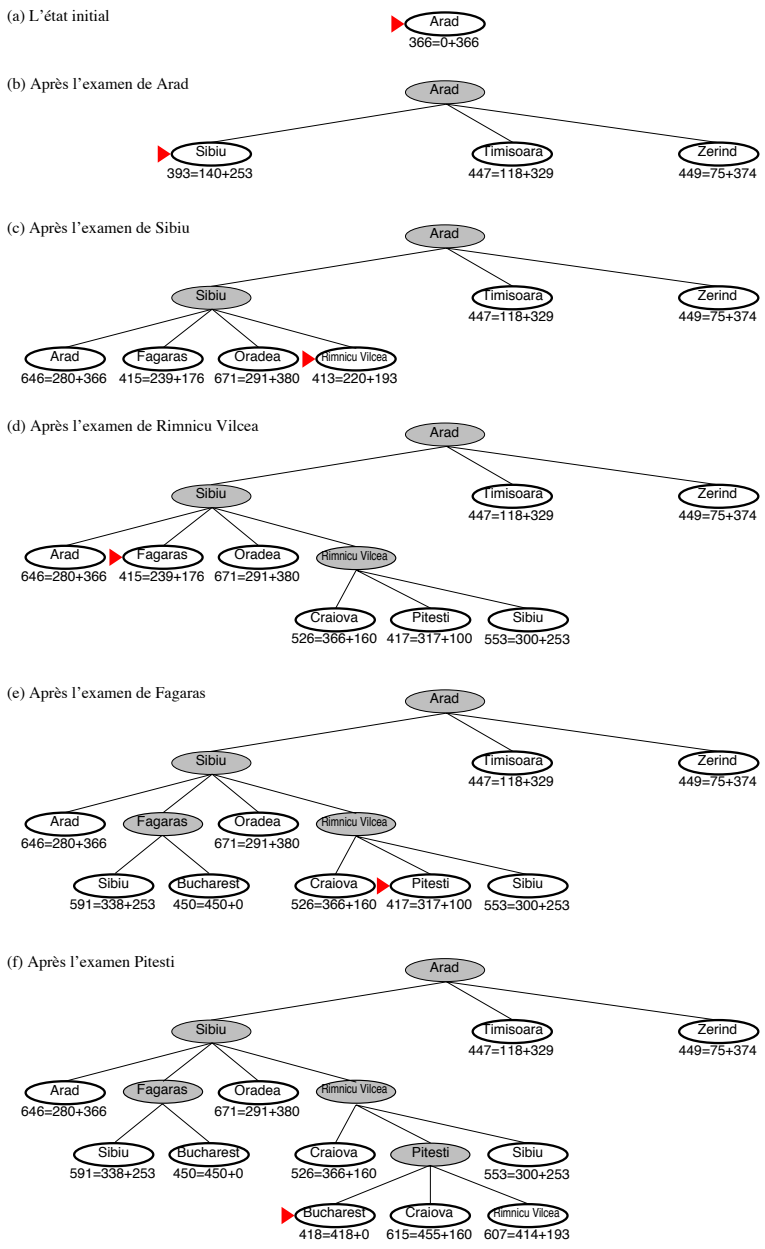

FIGURE 10 – La recherche A* pour trouver un chemin de Arad à Bucarest.

redondants (selon laquelle nous ne gardons que le premier chemin ammènant à un état), il nous faut placer une restriction plus forte sur la fonction heuristique h pour garantir l'optimalité : la fonction doit être *consistante*. Une fonction heuristique est consistante si pour tout noeud n et tout successeur n' obtenu à partir n en faisant l'action a, nous avons l'inégalité suivante :

$$h(n) \leq c(a) + h(n')$$

Les fonctions heuristiques consistantes sont toujours admissibles, mais les fonctions heuristiques admissibles sont très souvent, mais pas toujours, consistantes.

La complexité de la recherche A* dépend de la fonction heuristique en question. En général, la complexité en temps et en espace est grande, ce qui rend la recherche A* mal adapté pour les problèmes de grande taille. Pour pallier cet inconvénient, plusieurs autres algorithmes heuristiques moins gourmands en mémoire ont été proposés, mais nous ne les examinerons pas dans ce chapitre [5].

Recherche Locale

Nous avons vu que pour certains problèmes, en particulier les problèmes de satisfaction de contraintes, nous ne nous intéréssons par au chemin reliant l'état initial à l'état but mais seulement à l'état but lui-même. Pour les problèmes de ce type, il existe une autre stratégie possible : générer les états successivement sans s'intérésser aux chemins jusqu'à ce que nous trouvions un état but. Cette idée est à la base algorithmes de recherche locale, qui sacrifient la complétude pour gagner du temps et de la mémoire.

L'algorithme local le plus simple s'appelle la recherche locale gloutonne. Nous commençons avec un état choisi aléatoirement. Si l'état est un état but, nous arrêtons la recherche. Sinon nous générons ses successeurs et leurs valeurs heuristiques (voir Figure 11 (a)). S'il n'existe pas de successeur avec une meilleure valeur que la valeur heuristique de l'état actuel, nous pouvons plus améliorer la situation, donc nous arretons la recherche (voir Figure 11 (b)). Sinon, nous choissisons l'état successeur ayant la meilleure valeur heuristique, et nous continuons ainsi.

L'avantage de la recherche locale gloutonne est qu'elle a une très faible consommation en mémoire (il ne faut garder que l'état actuel en mémoire) et aussi en temps. Par exemple, pour le problème des huits reines, l'algorithme termine après 4 étapes quand il trouve une solution et 3 étapes quand

5. Pour ceux qui sont intéréssées, ces algorithmes s'appellent iterated A* search (IDA*), recursive best-first search (RBFS), memory-bounded A* (MA*), et simplified memory-bounded A* (SMA*).

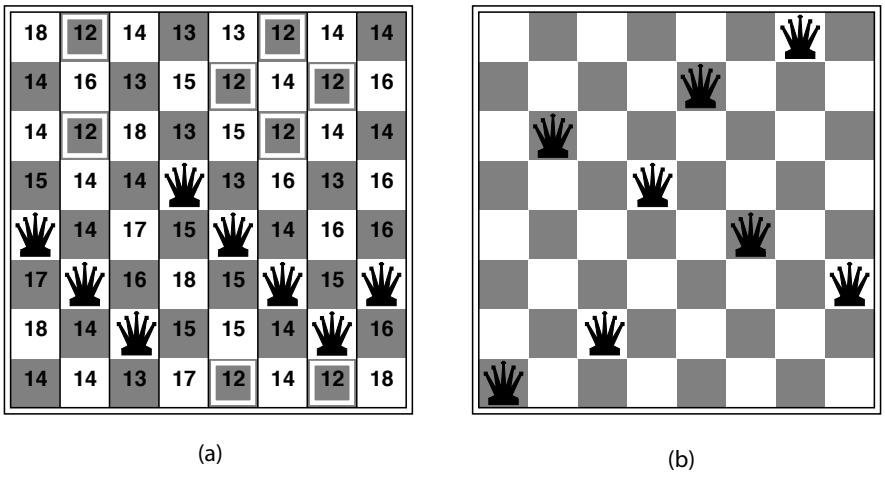

FIGURE 11 – (a) Les état successeurs sont obtenus en changeant la position d'une reine dans sa colonne. Les valeurs heuristiques (ici, le nombre de paires de reines qui s'attaquent) pour chaque état successeur sont presentées. La meilleure valeur est 12, donc un des états ayant une valeur 12 est séléctionné comme étant le prochain état. (b) Cet état a une valeur heuristique de 1, mais tous ses successeurs ont des valeurs ≥ 2, donc la recherche locale gloutonne s'arrête sur cet état.

il est bloqué (en moyenne). L'inconvenient principal de la recherche locale gloutonne est son faible taux de succès qui résulte du fait que nous arrêtons la recherche si jamais nous ne pouvons plus améliorer directement la valeur heuristique. Pour le problème des huit reines, cet algorithme ne réussit que 14% du temps.

Il y a plusieurs améliorations possible de la recherche locale gloutonne. Une idée simple serait de permettre l'algorithme de visiter les successeurs ayant la même valeur que l'état actuel (il faut mettre une borne sur le nombre de déplacements consécutives de ce type pour ne pas tourner en boucle). Cette modification augmente significativement le taux de succèss. Pour le problème des huits reines, si nous permettons au plus 100 déplacements consécutifs qui n'améliorent pas la valeur heuristique, le taux de succèss passe de 14% à 94%. Une autre possibilité est d'ajouter de l'aléatoire : au lieu de choisir toujours le meilleur voisin, nous pouvons choisir un voisin aléatoirement (où la probabilité de sélectionner un état est défini en fonction de sa valeur heuristique). Finalement, nous pouvons tout simplement recommencer la recherche locale gloutonne à partir d'un autre état choisi au hasard, et continuer ainsi jusqu'à obtenir un état but. Cette technique pourrait sembler un peu simpliste, mais elle s'avère très efficace : elle permet de résoudre le problème des 3 millions de reines en moins d'une minute sur un ordinateur de bureau !

Grace à leur très faible complexité en espace, les algorithmes locaux peuvent être utilisés pour résoudre des problèmes de taille importante qui ne peuvent pas être résolus avec des algorithmes classiques.

Chapitre 3: Planification

La planification, comme nous l'avons définie dans le dernier chapitre, consiste à produire une suite d'actions pour relier un état initial à un état but. Dans ce chapitre, nous allons voir en détail la spécification des problèmes de planification, et nous montrerons comment les algorithmes de recherche élaborés dans le chapitre précédent peuvent nous servir pour résoudre de tels problèmes. Nous considérerons également quelques autres approches de la planification et des extensions possibles du cadre classique de la planification.

Spécification des problèmes de planification

Dans le dernier chapitre, nous avons parlé brièvement de la spécification des problèmes de planification. En particulier, nous avons dit que les états correspondaient aux valuations de certaines propositions et que la fonction de successeur définissait quelles actions étaient possibles dans un état donné et comment les actions modifiaient l'état du monde. Dans cette section, nous introduisons un langage pour spécifier des problèmes de planification.

Le langage que nous allons utiliser s'appelle STRIPS. Nous allons voir abstraitement en quoi consiste ce langage, puis nous allons utiliser STRIPS pour formaliser des problèmes de planification concrets.

Etats du monde Nous allons définir les états des ensembles d'atomes. Des atomes peuvent être des propositions comme $Porte\_Ouverte$ ou des atomes complexes comme $Dans(lait, panier)$. Les atomes complexes sont construits à partir d'un ensemble de prédicats, e.g. $Dans$, et des constantes, e.g. $lait$, $panier$. Chaque prédicat a un nombre fixe d'arguments, e.g. $Dans$ a 2 arguments. Nous faisons la *supposition du monde clos*, c'est à dire que toutes les atomes qui ne sont pas dans l'état sont supposés faux. Cette supposition nous permet de représenter les états de façon plus compacte.

But Le but va être un ensemble d'atomes. Un état est un état but si tous les atomes du but sont vrais dans l'état, i.e. si le but est un sous-ensemble de l'état.

Description des actions Nous allons permettre les actions simples comme *dormir* mais aussi les actions prenant des arguments comme $marcher(laposte, supermarché)$. Pour décrire les actions, il nous faut spécifier leurs préconditions (les conditions nécessaires pour les exécuter) et leurs effets (comment ils affectent l'état du monde). En STRIPS, les préconditions sont simplement un ensemble d'atomes, et les effets sont décrits par deux ensembles, l'un donnant les atomes que l'action a rendu vrais, l'autre précisant les atomes que l'action a rendu faux. Pour ne pas avoir besoin d'écrire une description pour les différentes actions de type *marcher* ($marcher(maison, école)$, $marcher(laposte, maison)$, $marcher(maison, supermarché)$, etc.), ce qui serait redondant en plus d'être extrêmement pénible s'il y avait beaucoup de lieux différents, nous pouvons utiliser les variables dans les préconditions et les effets, à condition que ces variables soient des paramètres de l'action. Voici un exemple pour illustrer cela :

Action $marcher(l_1, l_2)$

Préconditions $Lieu(l_1), Lieu(l_2), Proche(l_1, l_2), \grave{A}(l_1)$

Effets $+\grave{A}(l_2)$ $-\grave{A}(l_1)$

Cette description indique que pour marcher de l_1 à l_2, il faut que l_1 et l_2 soient des lieux, que l_1 et l_2 soient proches, et que l'agent se trouve à l_1. A la fin de l'action, l'agent est maintenant à l_2 (ce que nous notons par $+\grave{A}(l_2)$) et ne se trouve plus à l_1 ($-\grave{A}(l_1)$). Nous remarquons que toutes les variables dans les préconditions et les effets font partie des paramètres de l'action– ce qui garantit que les préconditions et effets d'une action sont toujours des ensembles d'atomes (sans variables). Nous apellerons respectivement $Pre(A)$, $Pos(A)$, et $Neg(A)$ les préconditions, les effets postifis, et les effets négatifs d'une action A.

Une action A est exécutable dans l'état E_1 si ses préconditions sont satisfaites, i.e. $Pre(A) \subseteq E_1$. L'état résultant de l'exécution de A dans E_1 est égale à E_1 avec les effets positifs de A ajoutés et les effets négatifs de A supprimés, i.e. $E_2 = E_1 \cup Pos(A) \setminus Neg(A)$.

Nous allons maintenant montrer sur des exemples concrets comment représenter un problème de planification avec STRIPS. Nous commençons avec un exemple très simple où nous n'avons que des atomes et actions simples :

Exemple 1 (Pizza)**.** Dans cet exemple, le but de notre agent est d'être à la maison avec une pizza. Dans l'état initial, notre agent est à la maison avec de l'argent, son téléphone fonctionne, et il n'a pas encore passé une commande de pizza. Il peut passer une commande depuis la maison si son téléphone

fonctionne, ou il peut commander sa pizza directement dans le restaurant. L'agent peut aller au restaurant ou rentrer chez lui, et il peut prendre sa pizza s'il a déjà passé une commande et si il a de l'argent.

Atomes Nous utiliserons les atomes T, A, M, R, C, P respectivement pour le fait que le téléphone fonctionne, que l'agent ait de l'argent, que l'agent soit à la maison, que l'agent soit au restaurant, qu'une pizza soit commandée, et que l'agent ait sa pizza.

Etat initial L'état initial est $\{T, A, M\}$.

Etat but Le but est d'être à la maison (M) avec une pizza (P), ce qui donne $\{M, P\}$.

Actions Il y a 5 actions différentes : appeler pour passer une commande (*appeler*), aller au restaurant (*aller*), rentrer du restaurant (*rentrer*), commander une pizza dans le restaurant (*commander*), et acheter sa pizza (*acheter*). Voici les déscriptions des différentes actions :

Action *appeler* **Préconditions** M, T **Effets** $+C$

Action *aller* **Préconditions** M **Effets** $+R$ $-M$

Action *rentrer* **Préconditions** R **Effets** $+M$ $-R$

Action *commander* **Préconditions** R **Effets** $+C$

Action *acheter* **Préconditions** A, C, R **Effets** $+P$ $-A$ $-C$

Maintenant que nous avons compris les bases, essayons de formaliser un problème un peu plus complexe :

Exemple 2 (Livraison)**.** Dans cet exemple, nous travaillons pour une compagnie de livraison. Notre compagnie déssert les villes de Paris, Marseille, et Lyon, et nous disposons d'un avion et de trois camions, un par ville. Aujourd'hui nous avons deux livraisons à effectuer : il faut livrer un DVD à Paul, qui habite à Lyon et un livre à Marie qui habite à Marseille. Actuellement, ces deux objets, ainsi que notre camion de Paris, se trouvent à notre entrepôt de Paris. L'avion est à l'aéroport de Paris, et les deux autres camions sont aux aéroports de Lyon et de Marseille.

Prédicats Nous allons utiliser les prédicats unaires[1] *Objet*, *Camion*, et *Avion* qui servent à spécifier qu'une certaine chose est un objet, un camion, ou un avion. Nous utiliserons le prédicat unaire *Lieu* pour désigner les lieux spécifiques, et *Aeroport* pour les aéroports (notons que les aéroports seront aussi des lieux). Enfin, le prédicat unaire $Ville$

1. Les prédicats *unaires* prennent un seul argument, les prédicats *binaires* en prennent deux, et les prédicats *ternaires* en prennent trois. Il n'y a pas de mots spécifiques pour les prédicats prenants plus que trois arguments.

sera utilisé pour spécifier des noms de ville. Nous aurons également besoin de quelques prédicats binaires : $Dans\_Vehicule$ (qui dit qu'un certain objet est dans un certain camion ou avion), $Dans\_Ville$ (qui dit qu'un certain lieu se trouve dans une certaine ville), et $\grave{A}$ (qui dit qu'une certaine chose est à un certain lieu).

Noms d'objets Pour cet exemple, nous avons deux objets à livrer dvd_de_paul et $livre\_de\_marie$, trois camions à notre disposition $camion\_paris$, $camion\_lyon$, et $camion\_marseille$, un avion $notre\_avion$, six lieux différents $entrepôt$, $chez\_marie$, $chez\_paul$, $aeroport\_paris$, $aeroport\_lyon$, et $aeroport\_marseille$, et les villes $paris$, $lyon$, et $marseille$.

Etat initial Notre état initial va être l'ensemble suivant :

$\{Objet(dvd\_de\_paul), Objet(livre\_de\_marie), Camion(camion\_paris),$
$Camion(camion\_lyon), Camion(camion\_marseille), Avion(notre\_avion),$
$Ville(paris), Ville(lyon), Ville(marseille), Aeroport(aeroport\_paris),$
$Aeroport(aeroport\_lyon), Aeroport(aeroport\_marseille), Lieu(entrepôt),$
$Lieu(chez\_marie), Lieu(chez\_paul), Lieu(aeroport\_marseille),$
$Lieu(aeroport\_paris), Lieu(aeroport\_lyon), Dans\_Ville(entrepôt, paris),$
$Dans\_Ville(chez\_paul, lyon), Dans\_Ville(chez\_marie, marseille),$
$Dans\_Ville(aeroport\_paris, paris), Dans\_Ville(aeroport\_lyon, lyon),$
$Dans\_Ville(aeroport\_marseille, marseille), \grave{A}(dvd\_de\_paul, entrepôt),$
$\grave{A}(livre\_de\_marie, entrepôt), \grave{A}(camion\_paris, entrepôt), \grave{A}(notre\_avion,$
$aeroport\_paris), \grave{A}(camion\_lyon, aeroport\_lyon), \grave{A}(camion\_marseille,$
$aeroport\_marseille)\}$

Notre état comporte des atomes décrivant l'état initial, e.g. $\grave{A}(dvd\_de\_paul,$ $entrepôt)$, ainsi que les faits décrivant les types des différents constantes, e.g. $Aeroport(aeroport\_lyon)$, et la géographie, e.g. $dans\_ville(entrepôt,$ $paris)$ [2] Comme vous pouvez remarquer, écrire l'état initial est assez long, même pour notre tout petit exemple. Néanmoins, ce serait pire si nous n'avions pas fait la supposition du monde clos, parce que nous aurions eu besoin d'écrire tous les atomes qui sont vérifiés dans l'état initial *plus* tous les atomes qui ne le sont pas.

But Nous voulons que le DVD soit livré à Paul et que le livre soit livré à

2. Vous avez peut-être remarqué que les atomes décrivant les types et la géographie ne vont pas être affectés par les actions et seront donc inclus dans tous les états (ce qui n'est pas les cas pour d'autres atomes comme $\grave{A}(dvd\_de\_paul, entrepôt)$). Pour obtenir une représentation plus compacte des états, nous pourrions effectivement stocker les atomes "inchangeables" à part et n'utiliser que les atomes restants dans la descriptions des états, ce qui dans notre cas réduirait la taille de l'état initial de 30 atomes à 6 atomes.

Marie, ce qui donne le but suivant dans le langage STRIPS :

$$\{\grave{A}(dvd\_de\_paul, chez\_paul), \grave{A}(livre\_de\_marie, chez\_marie)\}$$

Description des actions Dans notre exemple, il y aura six types d'actions : mettre un objet dans un camion, mettre un objet dans un avion, enlever un objet d'un camion, enlever un objet d'un avion, conduire un camion d'un lieu à un autre dans la même ville, et déplacer un avion d'un aéroport à un autre aéroport. Pour chaque action, il nous faut spécifier ses paramètres, ses préconditions, et ses effets. Commençons d'abord par l'action de mettre un objet dans un camion :

Action $charger\_camion(o, c, l)$

Préconditions $Objet(o), Camion(c), Lieu(l), \grave{A}(o, l), \grave{A}(c, l)$

Effets $+Dans\_Vehicule(o, c)$ $-\grave{A}(o, l)$

Cette action prend trois paramètres, correspondant à l'objet, le camion, et le lieu où se trouve l'action. Les préconditions exigent que les trois paramètres sont bien typés (i.e. que o soit bien un objet, que c soit bien un camion, et que l soit bien un lieu) et que l'objet et le camion se trouvent au même endroit. Après l'exécution de $charger\_camion(o, c, l)$, l'objet o est dans le camion c (ce qui donne respectivement l'effet positif $+Dans\_Vehicule(o, c)$ et l'effet négatif $-\grave{A}(o, l)$).

Considérons ensuite l'action de décharger un objet d'un camion :

Action $décharger\_camion(o, c, l)$

Préconditions $Objet(o), Camion(c), Lieu(l), Dans\_Vehicule(o, c), \grave{A}(c, l)$

Effets $-Dans\_Vehicule(o, c)$ $+\grave{A}(o, l)$

Nous avons les trois mêmes paramètres que pour l'action précédent. Les préconditions exigent que les trois paramètres sont bien typés, que l'objet o est dans le camion c, et que le camion c se trouve à l. Il y a deux effets : l'objet o est maintenant à l et n'est plus dans le camion c.

Enfin, considérons l'action de conduire un camion d'un lieu à un autre :

Action $conduire\_camion(c, l_1, l_2, v)$

Préconditions $Camion(c), Lieu(l_1), Lieu(l_2), \grave{A}(c, l_1), Dans\_Ville(l_1, v), Dans\_Ville(l_2, v), Ville(v)$

Effets $+\grave{A}(c, l_2)$, $-\grave{A}(c, l_1)$

Cette action prend quatre paramètres : un camion, deux lieux, et une ville. Les préconditions disent que les paramètres sont bien typés, que le camion c se trouve au lieu l_1, et que les deux lieux se trouvent dans

la même ville. Après l'exécution de l'action, le camion est à l_2 et n'est plus à l_1.

Les trois autres actions ressemblent fortement aux trois actions que nous venons de voir. Voici leurs descriptions :

Action $charger\_avion(o, a, l)$

Préconditions $Objet(o), Avion(a), Lieu(l), \grave{A}(o, l), \grave{A}(a, l)$

Effets $-\grave{A}(o, l)\ +Dans\_Vehicule(o, a)$

Action $décharger\_avion(o, a, l)$

Préconditions $Objet(o), Avion(a), Lieu(l), Dans\_Vehicule(o, a), \grave{A}(a, l)$

Effets $+\grave{A}(o, l)\ -Dans\_Vehicule(o, a)$

Action $voler\_avion(a, l_1, l_2)$

Préconditions $Avion(a), \grave{A}(a, l_1), Aeroport(l_1), Aeroport(l_2)$

Effets $+\grave{A}(a, l_2),\ -\grave{A}(a, l_1)$

Dans cet exemple, nous n'avons eu que deux objets à livrer et trois villes désservies, mais notre formalisation pourrait être légèrement modifiée pour traiter d'autres exemples plus réalistes avec plus de villes, camions, avions, et objets à livrer (Exercice).

Planification avec les algorithmes de recherche

Nous allons considérer dans cette section deux façons de planifier en utilisant des algorithmes de recherche. La première consiste à commencer par l'état initial et à ajouter les actions les unes après les autres pour construire des plans. La deuxième possibilité est de commencer par l'état but et de construire les plans dans le sens inverse (la dernière action, puis l'avant dernière action, etc.).

Recherche en avant

Avec la recherche en avant, nous construisons les plans action par action jusqu'à ce que nous trouvions un état but. Par souci de complétude, nous donnons la formalisation comme un problème de recherche (bien que ce soit assez simple) :

Etats Les ensembles d'atomes.

Etat Initial L'état initial du problème.

Actions Les actions du problème.

Fonction de successeur Un état E_2 est un successeur de l'état E_1 s'il existe une action A telle que A est exécutable dans E_1 ($Pre(A) \subseteq E_1$) et E_2 est l'état résultant de l'exécution de A dans E_1 ($E_2 = E_1 \cup Pos(A) \setminus Neg(A)$).

Test de but Un état est un état but s'il contient toutes les atomes du but.

Coût des actions Chaque action coûte 1.

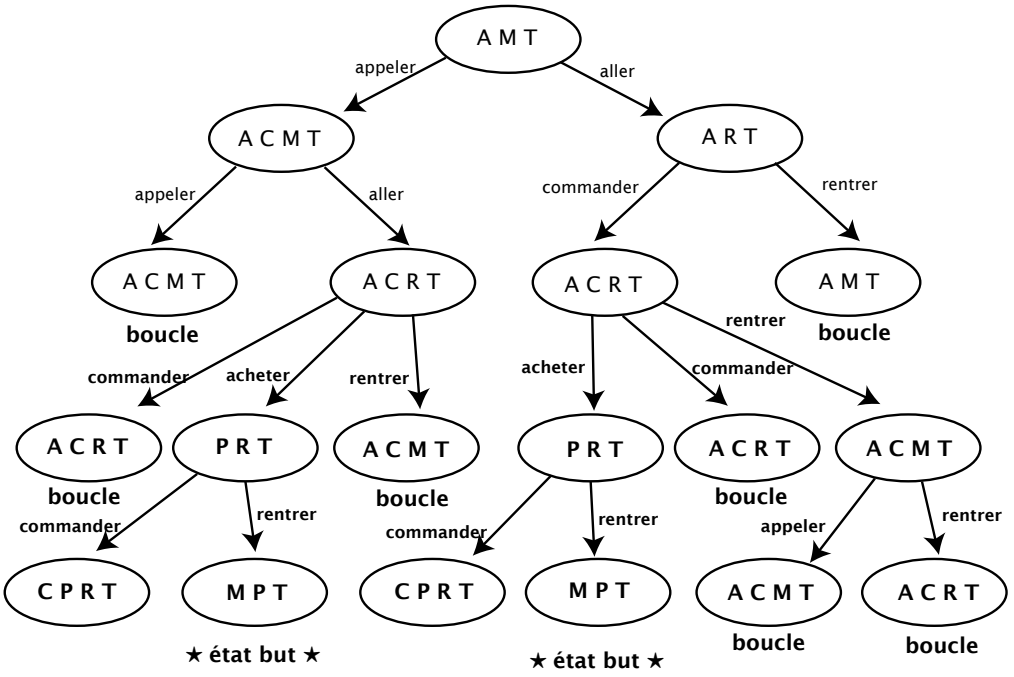

FIGURE 1 – L'espace de recherche (pour une recherche en avant) pour le problème de la pizza. Le premier noeud correspond à l'état initial du problème, et les états buts satisfont le but du problème.

Tous les algorithmes que nous avons vus dans le dernier chapitre - parcours en largeur, parcours en profondeur, A*, etc. - peuvent être utilisés pour trouver une solution à un problème de planification. Si nous voulons éviter de tourner en boucle, il faut garder en mémoire les différents états sur les chemins (et pas seulement le dernier état du chemin). Concrètement, cela veut dire que chaque noeud va contenir toutes les actions sur le chemin, comme

d'habitude, mais aussi tous les états rencontrés sur ce chemin. Nous pourrions également garder en mémoire la liste des états déjà visités pour éviter de considérer plusieurs fois les mêmes états.

Nous allons maintenant montrer comment les algorithmes de recherche peuvent être utilisés pour résoudre le problème de la pizza de l'Exemple 1. L'espace de recherche pour ce problème est présenté sur la Figure 1. La façon dont cet espace est exploré dependra bien sûr de l'algorithme de recherche en question. Nous considérons ici le parcours en largeur où les états redondants sont supprimés.Nous commençons avec un noeud $\{A, M, T\}$ correspondant à l'état initial, puis comme l'état initial n'est pas un état but (l'agent n'a pas sa pizza), nous produisons ses deux états successeurs : $\{A, C, M, T\}$ (correspondant à l'action *appeler*) et $\{A, R, T\}$ (correspondant à l'action *aller*). Nous avons donc deux noeuds à traiter : $\{A, M, T\}$-*appeler*-$\{A, C, M, T\}$ et $\{A, M, T\}$-*aller*-$\{A, C, R, T\}$. Comme $\{A, C, M, T\}$ et $\{A, C, R, T\}$ ne sont pas des états but non plus, nous continuons la recherche, produisant les noeuds associés aux deux succeseurs $\{A, C, M, T\}$ qui sont $\{A, C, R, T\}$ (obtenu en faisant l'action *aller*) et $\{A, C, M, T\}$ (par l'action *appeler*). Le dernier noeud est supprimé parce que son chemin contient deux fois l'état $\{A, C, M, T\}$. Nous calculons aussi les noeuds correspondant aux successeurs de $\{A, R, T\}$: $\{A, C, R, T\}$ (via l'action *commander*) et $\{A, M, T\}$ (avec l'action *rentrer*). Le noeud correspondant à $\{A, C, R, T\}$ est supprimé parce que nous avons déjà un noeud ammenant à $\{A, C, R, T\}$, et le noeud correspondant à $\{A, M, T\}$ est lui aussi supprimé de la liste de noeuds à traiter parce qu'il contient deux fois $\{A, M, T\}$. Ceci ne nous laisse qu'un seul noeud à traiter : $\{A, M, T\}$-*appeler*-$\{A, C, M, T\}$-*aller*-$\{A, C, R, T\}$. Comme le noeud ne correspond pas à un état but, nous créons ses trois successeurs qui correspondent aux états $\{A, C, R, T\}$ (l'action *commander*), $\{P, R, T\}$ (via *acheter*), et $\{A, C, M, T\}$ (l'action *rentrer*). Les premier et troisième noeuds sont supprimés parce qu'ils contiennent des boucles. Il ne reste que le noeud associé à $\{P, R, T\}$, qui a deux successeurs correspondant aux états $\{C, P, R, T\}$ (faisant *commander*) et $\{M, P, T\}$ (l'action *rentrer*). Comme $\{M, P, T\}$ est un état but, nous renvoyons le chemin associé : *appeler-aller-acheter-rentrer*. Notons que si nous avons traité les noeuds dans un ordre différent, nous aurions gardé le deuxième noeud correspondant à l'état $\{A, C, R, T\}$, ce qui nous aurait permis de trouver l'autre solution *aller-commander-acheter-rentrer*.

Les algorithmes de recherche non-informés sont généralement insuffisants pour résoudre des problèmes de planification d'une taille raisonnable. Prenez par exemple notre problème de livraison. Vous pouvez vérifier que dans ce problème la solution minimale comporte 17 actions, et il y a toujours au moins 5 actions exécutables dans un état donné. Il y a donc $> 5^{17}$ différents chemins

à examiner pour trouver une solution. Même si nous évitons les boucles et les états déjà visités, la taille de l'espace de recherche reste gigantesque. Et tout cela pour un problème où il n'y a que trois villes et deux articles à livrer !

La solution est bien sûr d'utiliser les algorithmes informés, mais pour cela, il nous faut trouver des heuristiques. Une heuristique toute simple consiste à compter le nombre d'atomes de but qui ne sont pas satisfaits. Cette heuristique est très facile à calculer et peut parfois améliorer un peu les performances, mais elle n'est pas suffisament précise pour être vraiment utile. Par contre, il existe aujourd'hui de très bonnes heuristiques qui permettent la résolution de problèmes de taille importante. Malheureusement, le fonctionnement de ces heuristiques est assez compliqué, et nous ne les traiterons pas ici. L'idée de ces algorithmes est de simplifier le problème de planification (généralement en enlevant les effets négatifs des actions) puis d'estimer le nombre d'actions nécessaires pour atteindre un état but dans le problème simplifié. Les valeurs heuristiques sont calculées à partir d'un objet nommé graphe de planification. Nous reviendrons brièvement sur ces graphes de planification dans la section dédiée aux autres méthodes pour la planification.

Une autre méthode pour réduire l'espace de recherche est de poser des contraintes sur les plans en cours de construction. L'idée est d'identifier dès que possible les plans stupides et de les éliminer. Plus concrètement, cela veut dire qu'avant d'ajouter un noeud à la liste de noeuds à traiter, nous regardons si le noeud satisfait les contraintes, et si ce n'est pas le cas, nous ne l'ajoutons pas. Pour notre exemple de livraison, une contrainte serait de ne pas décharger un objet à un lieu ou il a été chargé. Nous ne considérons donc pas les noeuds dont la suite d'actions contient $charger(o, c, l)$ puis ensuite $décharger(o, c, l)$. En général, les contraintes sont élaborées par les humains et non pas automatiquement, mais si nous voulons traiter beaucoup de problèmes concernant le même domaine (e.g. la livraison), cela en vaut la peine.

Recherche en arrière

Un inconvenient de la recherche en avant est que nous considérons toutes les actions exécutables dans un état donné, même si certaines actions ne contribuent en rien à l'obtention d'une solution. Ce problème devient de plus en plus gênant lorsque le nombre d'actions devient important. La recherche en arrière essaie de pallier ce problème en considérant seulement les actions qui peuvent contribuer à un but ou sous-but du problème. Voici une formalisation :

Etats Les ensembles d'atomes.

Etat Initial Le but du problème.

Actions Les actions du problème.

Fonction de successeur Pour définir les successeurs, nous avons besoin des notions d'actions pertinentes et d'actions consistantes. Une action A est dite *pertinente* pour l'état E s'il existe un atome de E qui est un effet positif de A, c'est à dire que A contribue à l'obtention de l'état E. Une action A est *consistante* pour l'état E si aucun des atomes de E n'est un effet négatif de A, ce qui veut dire que l'exécution de A peut potentiellement permettre d'atteindre l'état E. Les successeurs d'un état E sont tous les états E_s tel qu'il existe une action A telle que :

1. A est une action pertinente et consistante pour E
2. $E_s = E \setminus Pos(A) \cup Pre(A)$

Test de but Un état est un état but si tous ces atomes font partie de l'état initial

Coût des actions Chaque action coûte 1.

La formalisation est peut-être un peu compliquée, mais l'idée est plutôt simple. Supposons que nous voulions atteindre un état satisfaisant l'ensemble d'atomes dans le but B depuis l'état initial I. Si I satisfait déjà toutes les atomes de B, il n'y a rien à faire, mais si ce n'est pas le cas, nous considérons toutes les actions qui pourraient rendre vrai au moins un des atomes de B. Parmi ces actions, nous ne gardons que celles qui ne suppriment aucun des atomes de B (parce qu'une action qui supprime l'un des atomes ne pourra jamais produire un état qui satisfait toutes les atomes de B), et pour chacune entre elles, nous calculons les atomes nécessaires pour exécuter cette action pour arriver dans B. Maintenant nous avons un nouveau but qui est d'atteindre l'un de ces états. Nous continuons ainsi jusqu'à ce que nous arrivions à un état dont tous les atomes sont présents dans l'état initial I. La solution au problème de planification est donc la séquence d'actions sur le chemin de l'état but à l'état initial mais dans l'ordre inverse. Si la recherche en arrière ne vous paraît toujours pas très claire, ne vous inquietez pas, nous allons passer maintentant à un exemple concret.

Figure 2 montre l'espace de recherche pour la recherche en arrière pour le problème de pizza. Nous montrons comment résoudre ce problème avec le parcours en largeur avec états redondants supprimés. Le premier noeud correspond au but $\{M, P\}$. Pour calculer ses successeurs, il faut trouver toutes les actions pertinentes pour $\{M, P\}$, c'est-à-dire toutes les actions qui ont M ou P comme effet. Il y a donc deux actions pertinentes : *rentrer* (qui a M comme effet positif) et *acheter* (qui rend P vrai). Ces deux actions sont consistantes parce que M et P ne figurent pas parmi leurs effets négatifs.

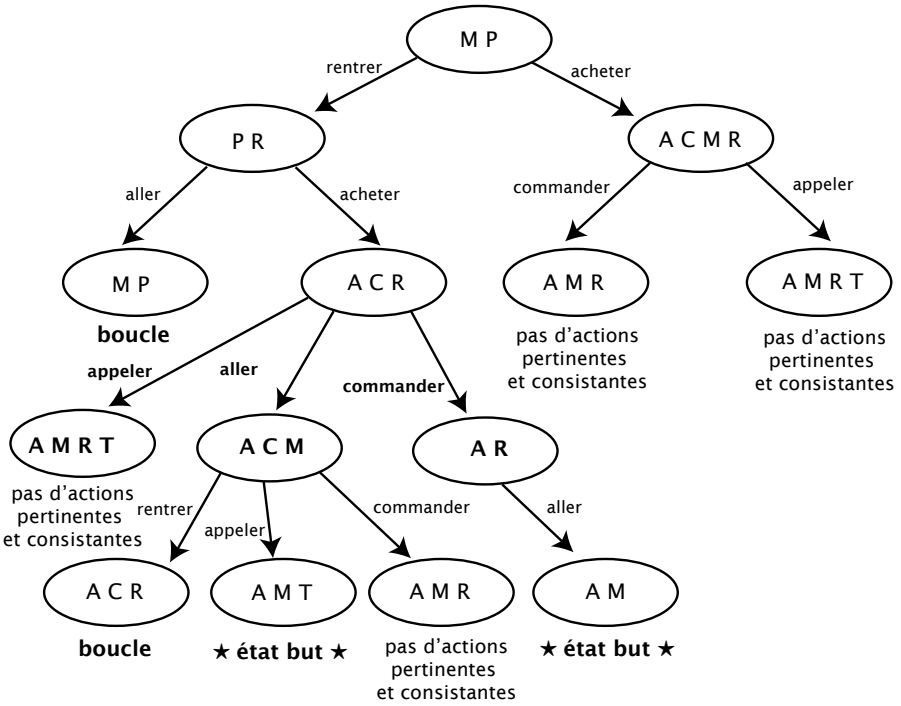

FIGURE 2 – L'espace de recherche (pour une recherche en arrière) pour le problème de pizza. Le premier noeud correspond au but du problème, et les "états but" sont des états dont toutes les atomes sont vrais dans l'état initial du problème.

Nous avons donc les deux noeuds $\{M, P\}$-*rentrer*-$\{R, P\}$ et $\{M, P\}$-*acheter*-$\{A, C, M, R\}$. Comme ni $\{R, P\}$ ni $\{A, C, M, R\}$ n'est satisfait dans l'état initial, nous poursuivons la recherche en calculant les actions pertinentes et consistantes pour ces deux états. Pour $\{R, P\}$, il y a deux actions pertinentes et consistantes : *aller* (qui rend R vrai) et *acheter* (qui rend P vrai). L'action *aller* ammène à l'état $\{M, R\}$ qui a été déjà visité, donc nous ne le gardons pas. L'action *acheter* donne l'état $\{A, C, R\}$ (parce que *acheter* produit P et nécessite A et C) qui n'a pas encore été considéré, donc nous ajoutons le noeud $\{M, P\}$-*rentrer*-$\{R, P\}$-*acheter*-$\{A, C, R\}$ à la liste de noeuds à traiter. Pour $\{A, C, M, R\}$, il y a deux actions pertinentes : *commander* qui donne l'état successeur $\{A, M, R\}$ et *appeler* qui donne l'état $\{A, M, R, T\}$. Ces deux états n'ont pas encore été considérés, donc nous ajoutons les noeuds

correspondants à la liste de noeuds à traiter, qui compte maintenant trois noeuds. Nous considérons ensuite le noeud dont l'état est $\{A, C, R\}$ dont les actions pertinentes sont *appeler*, *aller*, et *commander*. Ces actions sont aussi consistantes avec $\{A, C, R\}$ parce qu'elles ne comptent aucun des atomes de $\{A, C, R\}$ parmi leurs effets négatifs. Nous ajoutons les noeuds pour les actions *aller* (qui donne l'état successeur $\{A, C, M\}$) et *commander* (qui donne A, R) mais pas pour *appeler* puisque l'état successeur ($\{A, M, R, T\}$) est redondant. Maintenant, nous considérons le noeud dont l'état est $\{A, M, R\}$, et nous trouvons ses actions pertinentes : *aller* (qui rend M vrai) et *rentrer* (qui rend M vrai). Mais ces actions ne sont pas consistantes avec $\{A, M, R, T\}$ parce qu'elles ont des atomes de $\{A, M, R, T\}$ sur leurs listes d'effets négatifs (*aller* supprime M, *rentrer* supprime R). Il n'y a donc pas de successeurs pour ce noeud, et vous pouvez verifier que $\{A, M, R, T\}$ n'a pas de successeurs non plus. Il nous reste donc les deux noeuds correspondant respectivement à l'état $\{A, C, M\}$ et l'état $\{A, R\}$. Pour $\{A, C, M\}$, il y a trois actions pertinentes et consistantes : *rentrer*, *appeler*, *commander*. Les états successeurs pour *rentrer* et *commander* ont été déjà visités lors de la recherche, donc il n'y aura qu'un seul noeud à ajouter, celui qui correspond à l'état but $\{A, M, T\}$. La solution trouvée est la suite d'actions associée à ce noeud : *rentrer-acheter-aller-appeler*, ce qui donne le plan *appeler-aller-acheter-rentrer*. Si nous avions considéré le noeud de l'état $\{A, R\}$, nous aurions trouvé l'autre solution.

La recherche en arrière s'avère souvent plus efficace que la recherche en avant, parce qu'elle est plus ciblée. Néanmoins, la recherche en arrière souffre d'autres inconvénients, notamment le fait que les états produits lors de la recherche ne sont pas forcément accessibles depuis l'état initial du problème de planification. Prenons par exemple l'ensemble d'atomes $\{A, M, R\}$ dans lequel l'agent est à la fois à la maison et au restaurant. Nous pourrions jamais atteindre un état satisfaisant $\{A, M, R\}$ depuis l'état initial parce que lorsque l'agent va au restaurant, il est plus chez lui, et vice-versa. Il est donc inutile de considérer de tels états parce qu'ils ne pourront jamais contribuer à une solution. En général, il est trop couteux de caractériser complètement l'ensemble des états accessibles depuis l'état initial, mais parfois nous pouvons identifier une partie de ces états (e.g. tous les états contenant M et R) et réduire ainsi la taille de l'espace de recherche. Notons également que les heuristiques et les contraintes dont nous avons parlé dans la dernière section peuvent aussi être utilisées pour la recherche en arrière.

D'autres algorithmes de planification

La recherche dans l'espace des états constituent une façon de planifier, mais ce n'est pas la seule. Nous considérons brièvement quelques d'autres méthodes :

Graphplan L'algorithme Graphplan construit incrémentalement un graphe qui comporte des couches successives de noeuds d'atomes et d'actions. La première couche correspond aux atomes de l'état initial, la deuxième couche comport toutes les actions qui peuvent être exécutées dans l'état initial, la troisième couche contient tous les atomes présents dans la première couche plus tous les atomes qui pourraient devenir vrais grâce à une des actions dans la deuxième couche, et ainsi de suite. Nous mettons un arc d'un noeud atome vers un noeud action si l'atome figure dans les préconditions de l'action, et nous mettons un arc d'un noeud action vers un noeud atome si l'action a l'atome comme un effet positif. Enfin, pour chaque couche d'atomes ou d'actions, nous étiquetons les paires d'actions et des paires d'atomes qui sont en conflit (e.g. une action est en conflit avec une autre action si la première supprime une précondition de la deuxième). Nous arrêtons la construction du graphe quand nous trouvons deux couches d'atomes successives identiques (dans ce cas, ce n'est pas la peine de continuer comme nous ne trouverons plus rien de nouveau) ou quand nous avons trouvé une couche dans laquelle toutes les atomes du but apparaissent sans conflit (il y a peut-être une solution). Dans le premier cas, il n'y a pas de solution, donc nous arrêtons l'algorithme, et dans le deuxième, nous faisons une recherche dans le graphe pour essayer de trouver une solution. L'algorithme Graphplan n'est plus parmi les plus performants aujourd'hui. En revanche le graphe de planification a eu un impact considérable dans le domaine : la plupart des bonnes heuristiques actuelles ont été obtenues à partir d'un examen du graphe de planification, et ce graphe est aussi utilisé pour encoder des problèmes de planification en des problèmes de satisfiabilité ou de satisfaction de contraintes.

Traduction en un problème de satisfiabilité [3] **ou en un problème de satisfaction de contraintes** Ces algorithmes comportent trois étapes : tout d'abord nous codons notre problème de planification en un problème de satisfiabilité (SAT) ou de satisfaction de contraintes (CSP), puis nous utlisons un solveur SAT ou CSP pour trouver une solution (s'il y en a

3. Un problème de satisfiabilité est de trouver une valuation des variables propositionelles qui satisfait une certaine formule (qui est donnée en entrée). Par exemple, la valuation $a = vrai, b = faux$ serait une valuation qui satisfait la formule $a \wedge \neg b$. Les problèmes de satisfiabilité sont des cas particuliers des problèmes de satisfaction de contraintes.

une), et enfin nous traduisons la solution ainsi obtenue en une solution pour notre problème de planification. Un des avantages de ce type de planificateur est que nous pouvons exploiter les progrès faits sur les solveurs SAT ou CSP : quand un nouveau solveur SAT ou CSP plus performant est developpé, nous pouvons l'utiliser dans notre planificateur obtenant ainsi de meilleures performances sans avoir fait de réels changements sur notre système.

Planification d'ordre partiel Les plans que nous générons avec la recherche dans l'espace des états sont des plans totalement ordonnés (ou séquentiels), c'est-à-dire que l'ordre des actions est totalement détérminé. La planification d'ordre partiel construit des plans partiels où l'ordre des actions n'est pas forcément complètement détérminé. Plus concrètement, un plan partiel est défini comme un ensemble d'actions plus des contraintes sur l'ordre de ces actions (e.g. $aller < rentrer$). Figure 3 montre un plan partiel pour un problème très simple de livraison, puis les six plans séquentiels qui lui correspondent. Travailler avec les plans partiels permet donc de réduire l'espace de recherche parce qu'un plan partiel représente plusieurs plans séquentiels (qui sont traités séparément par les algorithmes de recherche dans l'espace des états). Les plans partiles sont aussi plus flexibles (comme il nous laisse le choix d'ordonner les actions non-ordonnées comme nous voulons) et plus naturels. Des plans partiels sont construits petit à petit, ajoutant à chaque étape soit une action soit une contrainte sur l'ordre des actions.

Notons que même si ces autres approches ne font pas une recherche sur l'espace des états du monde, ils font tous appel aux algorithmes de recherche :

- → la recherche est utilisée pour extraire les solutions d'un graphe de planification
- → après avoir traduit notre problème en un problème SAT ou CSP, nous appelons un solveur SAT ou CSP, qui lui est basé sur un algorithme de recherche
- → pour générer les plans partiels, nous faisons une recherche dans l'espace des plans partiels

Limitations de notre formalisation

Le langage STRIPS que nous avons introduit dans ce chapitre a été le standard pendant plus de quinze ans et a eu une grande influence sur la recherche dans le domaine de la planification. Néanmoins, STRIPS reste un langage très simple qui n'est pas suffisament expressif pour nombre d'applications pratiques. Dans cette section, nous allons revenir sur cette formali-

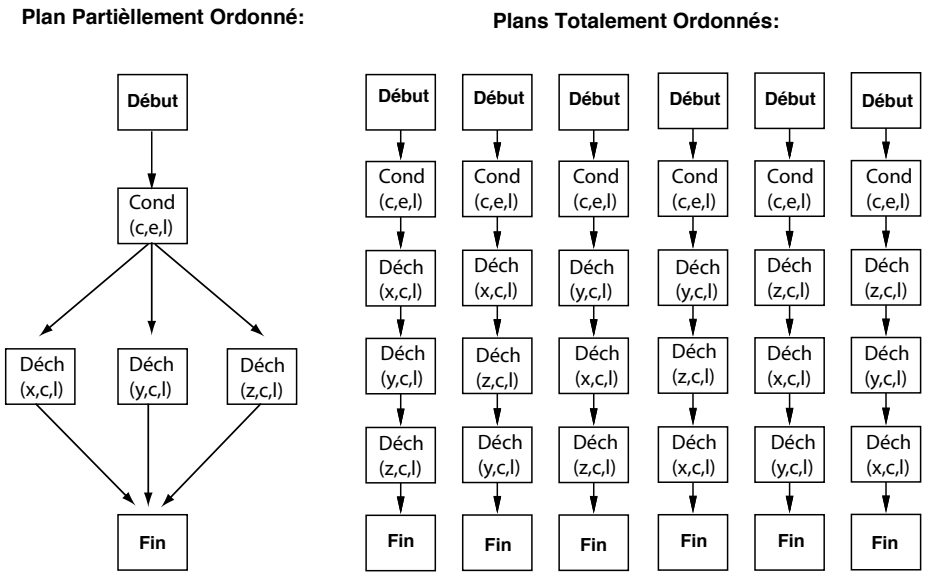

FIGURE 3 – Un plan partiellement ordonné et les six plans totalement ordonnés qui lui correspondent.

sation, identifier les restrictions que nous avons faites, et voir comment nous pourrions les affaiblir.

Définition des actions et leurs effets Les actions, comme nous les avons définies dans ce chapitre, ont toujours le même effet, sont instantanées, ne consomment pas de ressources, et n'ont que des effets physiques.

Effets conditionnels Rappelons l'action *actionner_interrupteur* qui nous avons vu lors du dernier chapitre qui allume la lumière si la lumière n'est pas allumée et éteint la lumière dans le cas contraire. Il n'y a pas de façon simple de décrire cette action dans le langage STRIPS parce que son effet dépend de l'état dans laquelle l'action est exécutée. En fait, il serait nécessaire de décomposer cette action en deux autres actions, l'une qui est exécutable quand la lumière est allumée et l'autre qui est seulement exécutable quand la lumière est éteinte. D'autres formalismes plus expressifs permettent des effets conditionnels, simplifiant

ainsi l'écriture de telles actions.

Actions non-détérministes Toutes les actions que nous avons considérées jusqu'à maintenant sont détérministes, c'est à dire qu'elles ne donnent lieu qu'à un seul état successeur. Cette supposition est raisonnable pour de nombreux problèmes, mais n'est pas toujours vérifiée, en particulier quand les actions peuvent échouer. Notons également que le non-détérminisme peut également provenir d'un manque de connaissance de l'agent, dans le cas par exemple où l'agent ne peut pas prédire l'état résultant de l'exécution d'une action (même une action détérministe dont il connaît les effets) s'il ne connaît pas l'état dans lequel il se trouve. Depuis 2004, une partie de la compétition internationale de planification est dédiée à la planification dans les domaines non-déterministes, ce qui témoigne de l'intérêt grandissant pour ce type de problème.

Actions temporelles Les actions que nous avons considérées ne sont pas fixées dans le temps. Pour beaucoup de problèmes dans le monde réel (y compris le problème de livraison examiné plus haut), nous avons besoin de plus de précisions : nous voulons savoir à quels moments les actions devraient être exécutées et combien de temps chaque action devrait durer. Une façon de résoundre de tels problèmes est de scinder le problème en deux : nous contruisons d'abord un plan puis nous nous occupons de l'ordonnancement des différentes actions. En général, nous souhaitons ordonnancer les actions pour que le temps total de l'exécution du plan soit le plus petit possible. Les planificateurs produisant des plan partiels sont donc mieux adaptés car ils spécifient les actions qui peuvent être exécutées en paralèlle.

Actions dépendant ou affectant les états mentaux Dans notre formalisation, les préconditions et les effets des actions ne concernaient que l'état du monde. Il existe cependant des actions dont l'exécution dépend de l'état de connaissance de l'agent, e.g. pour consulter ses mails, il faut connaitre son identifiant et son mot de passe. D'autres actions pourraient fournir des informations aux agents, e.g. après avoir consulté les pages jaunes, l'agent connait le numéro de téléphone de son ami. Incorporer de telles actions nécessite des modifications non-triviales de notre formalisme.

Actions menées par plusieurs agents Dans notre formalisation, toutes les actions sont exécutées par un seul agent, mais nous pourrions bien imaginer les situations où il serait nécessaire de coordonner les actions de plusieurs agents afin de réaliser un but commun. Ce problème, connu sous le nom de "multi-agent planning", est bien plus complexe que la planification mono-agent parce qu'il faut que les agents se mettent d'accord sur un plan commun.

Définition d'un but : Dans notre formalisation, il n'y a qu'un seul but qui définit les caractéristiques souhaitées de l'état final du plan. Le but est soit vrai, soit faux dans un état donné.

Ensemble de buts : Au lieu de ne définir qu'un seul but, nous définissons un ensemble de buts, et nous cherchons un plan qui satisfait le plus grand nombre de buts. Pour aller encore plus loin, nous pouvons associer un poids à chaque but (selon son importance) et essayer de trouver des plans dont la somme des poids des buts satisfaits est maximal. Ce type de planification est connu sous le nom de "oversubscription planning".

Buts temporels : Dans notre formalisation, le but ne concerne que l'état final d'un plan, mais souvent il semble utile d'avoir des buts qui prennent en compte toutes les actions et les états d'un plan. Par exemple, on pourrait demander qu'une certaine propriété reste vraie pendant l'exécution du plan, ou bien que certaines actions soient exécutés dans le plan dans un ordre particulier. Ces buts plus généraux sont appellés "temporally extended goals".

Plans Préférés : Un but sépare l'ensemble des plans en deux catégories : ceux qui satisfont le but, et ceux qui le satisfont pas. Cette formalisation ne nous permet pas de distinguer les différentes façons d'atteindre le but. L'idée serait alors d'ajouter au but des préférences de l'utilisateur et de chercher des plans qui satisfont le but *et* satisfont le plus possible des préférences. Pour prendre un exemple simple, quand nous planifions un voyage, il y a généralement plusieurs vols possibles, mais nous avons souvent des préferences sur le prix du billet, la compagnie aérienne, le nombre de connections, etc. En 2006, les préférences simples ont été ajoutées au langage utilisé pour la compétition internationale de planification.

Définition d'une solution Nous avons défini les solutions comme des suites d'actions qui relient l'état initial à un état but. Dans la dernière section, nous avons déjà vu quelques autres définitions de solutions (les plans partiels, par exemple). Voici une dernière définition de solutions qui est intéressante dans le cadre de la planification avec des actions non-déterministes :

Plan conditionnel Comment planifier quand nous ne savons pas quels seront les résultats des actions ? Si nous connaissons quand même les états possibles après chaque action, nous pourrions envisager de créer un plan pour chacune des possibilités, puis quand nous exécutons ce plan conditionnel, nous allons vérifier lequel des états possibles est l'état résultant et nous allons suivre le plan correspondant. Par exemple, nous pourrions avoir le plan conditionnel pour comment passer l'après-midi :

si pluie **alors** [**si** motivé **alors** travailler **sinon** aller sur l'internet]
sinon se ballader dehors

Chapitre 4: Jeux

Dans ce chapitre, nous allons voir comment la recherche peut être utilisée dans les jeux. Nous commençons par une description et une formalisation d'une certaine classe de jeux. Puis nous allons introduire des algorithmes qui permettent d'obtenir les stratégies optimales pour cette classe de jeux. Comme le temps de calcul nécessité par ces algorithmes les rend inutilisablee en pratique, nous consacrons la dernière partie du chapitre aux modifications possibles afin de pouvoir prendre les décisions en temps réel.

Formalisation des Jeux

Il existe de nombreux types de jeux avec des propriétés très différentes. En intelligence artificielle, la plupart de la recherche s'est focalisée autour de la classe de jeux finis déterministes à deux joueurs, avec tours alternés, à information complète, et à somme nulle. Examinons chacune de ces caractéristiques :

Fini Un jeu est *fini* si le jeu termine toujours. Dans beaucoup de jeux, e.g. les échecs, les règles du jeu imposent des conditions pour rendre le jeu fini.

Détérministe Un jeu est *déterministe* si le déroulement du jeu est entièrement déterminé par les choix des joueurs. Des jeux utilisant les dés (e.g. le Monopoly, le backgammon) ou des cartes distribuées au hasard (e.g. le poker,) sont *non-déterministes*.

Information complète Dans les jeux à *information complète*, les joueurs connaissent parfaitement la configuration actuelle du jeu. Ce n'est pas le cas, par exemple, avec la plupart des jeux de cartes (e.g. le poker) où les joueurs ne connaissent pas les cartes des autres. Ce sont des jeux à *information partielle*.

Somme nulle Un jeu est dit à *somme nulle* si la valeur gagnée par l'un des joueurs est la valeur perdue par l'autre. Le poker est un exemple d'un jeu à somme nulle parce que le montant qui est gagné par un joueur

est égal à la somme des montants perdus par les autres joueurs. Les jeux où il y a toujours un gagnant et un perdant sont des jeux à somme nulle.

Le morpion est un bon exemple d'un jeu satisfaisant ces critères. Il est fini parce qu'une partie dure au plus 9 tours, il est déterministe parce que le hasard n'invervient pas, il est à information complète parce que les deux joueurs connaissent l'état de la grille, et il est à somme nulle parce que lorsqu'un joueur gagne, c'est l'autre qui perd. Les échecs et le jeu de dames sont deux autres exemples de jeux de cette classe.

Passons maintenant à une formalisation des jeux de ce type :

Etat initial C'est une configuration du jeu plus le nom de joueur à jouer.

Fonction de successeur La fonction définit quelles actions sont possibles pour un joueur dans une configuration donnée, ainsi que les configurations résultantes des différentes actions.

Test de terminalité Ce test définit les configurations terminales du jeu.

Fonction d'utilité Cette fonction associe une valeur à chaque configuration terminale. Souvent, les valeurs sont +1, -1, et 0 qui correspondent à un gain pour le premier joueur, une perte pour le premier joueur, et un match nul (parfois $+\inf$ et $-\inf$ sont utilisés au lieu de $+1$ et -1). Notez que comme nous ne considérons que les jeux à somme nulle, nous n'avons besoin que d'une seule valeur par configuration : la valeur attribuée au deuxième joueur est exactement le négatif de la valeur reçue par le premier joueur.

L'objectif du jeu pour un joueur est de maximiser son gain. Plus spécifiquement, le premier joueur veut que la valeur d'utilité de la configuration terminale du jeu soit la plus grande possible, et inversement, le deuxième joueur souhaite que cette valeur soit la plus petite possible (parce que lui reçoit le négatif de cette valeur). Afin de pourvoir se rappeler de leurs objectifs respectifs, il est courant d'appeller les deux joueurs par `max` et `min`.

Essayons de formaliser ainsi le jeu de morpion :

Etat initial La grille vide et un nom de joueur à jouer, X par exemple.

Fonction de successeur Quand c'est le tour de joueur X, il peut mettre un X dans un des cases vides de la grille. Même chose pour O.

Test de terminalité Une configuration est terminale s'il y a trois X (ou trois O) sur une ligne, une colonne, ou une diagonale, ou si la grille est complète.

Fonction d'utilité Nous donnons une valeur de +1 à toute configuration terminale contenant trois X dans une ligne, colonne, ou diagonale (c'est

X qui gagne), et -1 à toute configuration terminale avec trois O dans une même ligne, colonne, ou diagonale (c'est O qui gagne). Les autres configurations terminales ont pour valeur 0, puisqu'elles correspondent à un match nul.

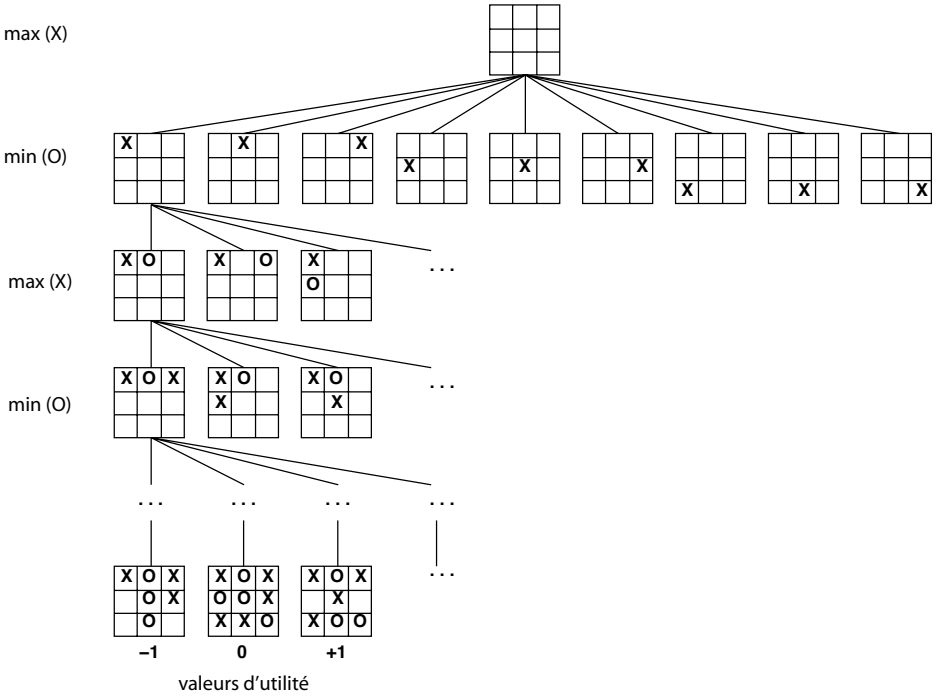

FIGURE 1 – Une partie de l'arbre du jeu pour le morpion.

Un jeu peut être réprésenté par un *arbre de jeu*, où les noeuds sont les configurations et les arcs les coups possibles. Les feuilles de l'arbre sont des configurations terminales, qui sont étiquetées par leurs valeurs d'utilité. Un noeud est appellé un noeud `max` si c'est à `max` de jouer, et est appellé un noeud `min` dans le cas contraire. Dans Figure 1, nous montrons une partie de l'arbre de jeu pour le morpion. La racine de l'arbre correspond à la configuration initiale dans laquelle la grille est vide. Les fils de la racine correspondent aux neuf coups possibles pour le joueur `max` (qui joue les X). Chacun de ces noeuds possède 8 fils, qui correspondent aux différentes façons pour `min` de

répondre, et ainsi suite. Trois des configuations terminales sont présentées : dans le premier, c'est `min` qui gagne (valeur d'utilité = -1), dans le deuxième, c'est un match nul (valeur 0), et dans le dernier, c'est une victoire pour `max` (valeur +1).

Pour les problèmes de recherche, nous cherchons une solution, c'est à dire une suite d'actions reliant l'état initial à l'état but. Pour les jeux, une simple suite de coups n'est pas suffisante pour jouer parce que nous ne savons pas quels seront les coups de l'adversiare. Il faut donc trouver une *stratégie* qui définit comment le joueur doit jouer dans toutes les évolutions possibles du jeu. Plus concrètement, une stratégie consiste en un premier coup, puis un choix de coup pour chaque réponse possible de l'adversaire, puis un choix de coup pour les réponses à notre dernier coup, et ainsi suite. On dit qu'une stratégie est *optimale* s'il n'existe pas d'autre stratégie qui donnerait un meilleur résultat contre un adversaire idéal.

Génération de Stratégies Optimales

Dans cette section, nous introduisons des algorithmes pour déterminer des stratégies optimales.

Algorithme Minimax

Une stratégie optimale peut être obtenue en examinant les *valeurs minimax* des noeuds de l'arbre de jeux. La valeur minimax d'un noeud donne la valeur d'utilité qui sera atteinte si chaque joueur joue de façon optimale à partir de la configuration de ce noeud. Pour calculer ces valeurs, nous commençons avec les configurations terminales, dont les valeurs minimax sont leurs valeurs d'utilité. Puis, nous évaluons chaque noeud non-terminal en utilisant des valeurs minimax de ces fils. La valeur minimax d'un noeud `max` sera le maximum des valeurs minimax de ces fils, parce que le joueur `max` veut maximiser la valeur de la configuration terminale. Pour les noeuds `min`, c'est l'inverse : nous prenons le minimum des valeurs minimax de ces fils car le joueur `min` cherche à obtenir une valeur minimale pour la configuration terminale. Nous pouvons l'exprimer ainsi :

$$\textsc{Minimax}(n) = \begin{cases} \textsc{Utilité}(n) & \text{si } n \text{ est terminal} \\ max_{s \in Successeurs(n)} \ \textsc{Minmax}(s) & \text{si } n \text{ est un noeud } \texttt{max} \\ min_{s \in Successeurs(n)} \ \textsc{Minmax}(s) & \text{si } n \text{ est un noeud } \texttt{min} \end{cases}$$

Nous illustrons le calcul des valeurs minimax sur l'arbre de jeux dans

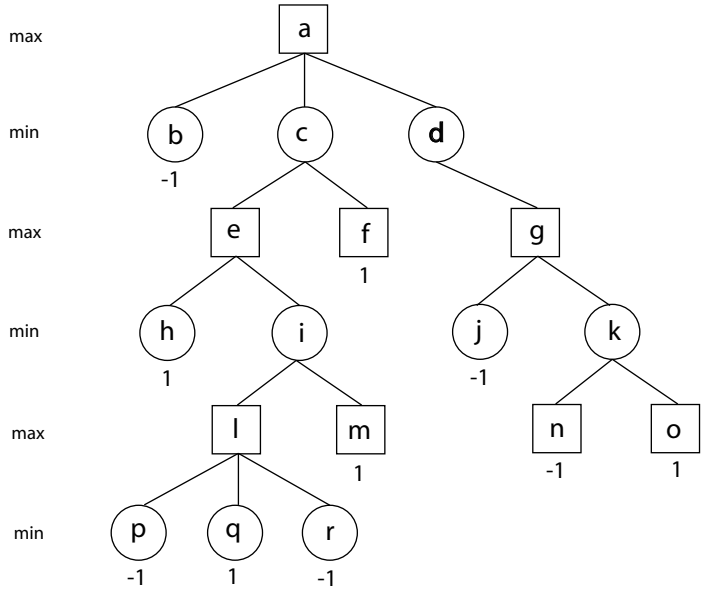

FIGURE 2 – Un exemple d'un arbre du jeu pour un jeu simple.

Figure 2 [1]. Les valeurs minimax des noeuds b, f, h, j, m, n, o, p, q, et r sont simplement leurs valeurs d'utilité, donc nous n'avons rien à calculer. Passons ensuite au noeud l. Ce noeud est un noeud `max` donc il faut prendre le maximum de ces fils, ce qui donne une valeur de 1. Pour le noeud i, qui est de type `min`, nous prenons le minimum de ces fils, obtenant ainsi une valeur de 1. Pour e, nous obtiendrons une valeur de 1 en prenant le maximum des valeurs de h et i. Pour c, nous avons une valeur de 1 également car les deux fils ont tous les deux une valeur de 1. Passons ensuite au noeud k qui est de type `min`. Sa valeur est -1 car nous prenons le minimum des valeurs de n et o. Nous avons une valeur de -1 aussi pour g (c'est le maximum des valeurs de j et k) et pour d (parce qu'il n'y a qu'un seul coup possible dans d). La valeur de a est donc 1 car c'est le maximum des valeurs de b, c, et d.

Comment déterminer la stratégie optimale à partir des valeurs minimax ? C'est très simple. Si nous sommes le joueur `max` et c'est à nous de jouer, nous devrons choisir un coup qui ammène à un noeud avec une valeur minimax maximale. Inversement, pour le joueur `min`, le meilleur coup est celui qui a la plus petite valeur minimax. Dans notre exemple, la valeur de a est 1, ce qui signifie que le joueur `max` peut gagner s'il suit la stratégie optimale, peu

1. Il serait évidemment préférable de travailler sur un vrai jeu, mais cela n'est pas vraiment possible car les arbres de jeux sont énormes même pour des jeux aussi simples que le morpion

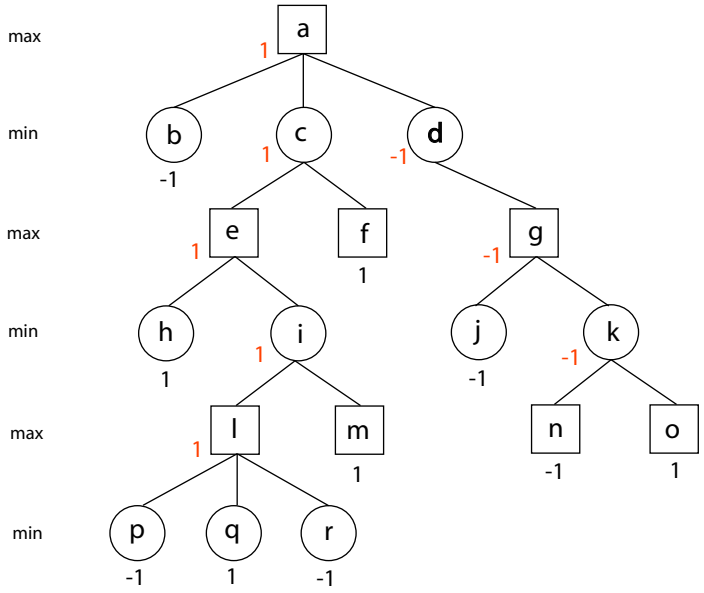

FIGURE 3 – L'algorithme minimax sur notre premier exemple.

importe les coups de l'autre joueur ! Quel est le meilleur premier coup pour `max` ? Nous voyons que c'est en allant vers c que nous arriverons à une valeur de 1. Si `min` choisit ensuite f, `max` gagne, et sinon `max` peut jouer h (et gagne toute de suite) ou peut jouer i et puis rien (si `min` joue m) ou q (si `min` choisit l). Dans tous les cas, c'est le joueur `max` qui gagne – à condition qu'il suive bien la stratégie donnée par minimax.

En principe, le calcul des valeurs minimax pourrait se faire en parcours en profondeur ou en parcours en largeur. Il est néanmoins préférable d'utiliser le parcours en profondeur car nous réduisons ainsi la complexité spatiale du calcul. Voici en plus de détailles l'algorithme récursif pour calculer les valeurs minimax (qui est une implémentation directe de la formule ci-dessous) :

fonction MINIMAX(n)
  si n est un noeud terminal **alors**
    RETOURNER la valeur d'utilité de n
  sinon
    si n est un nœud de type `min` **alors**
      $v = +\infty$
      pour tout fils f de n **faire**
        v = MIN(MINIMAX(f), v)

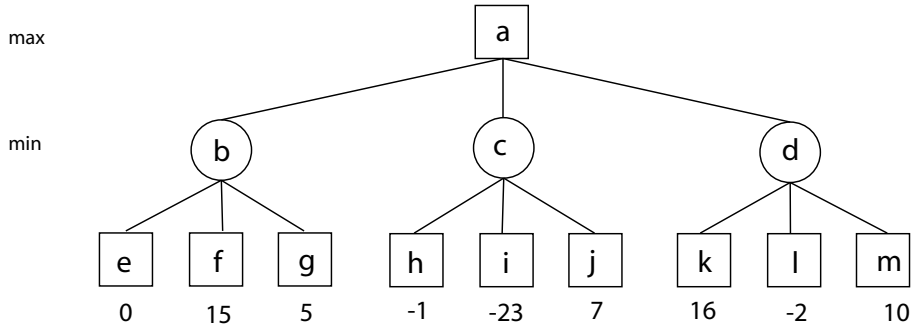

FIGURE 4 – Un autre exemple d'un arbre du jeu.

```
        RETOURNER v
      sinon
        v = −∞
        pour tout fils f de n faire
          v = MAX(MINIMAX(f), v)
         fin pour
        RETOURNER v
      fin si
    fin si
fin
```

Pour un arbre avec un facteur de branchement de b et une profondeur p, cet algorithme a une complexité spatiale de $O(p * b)$ (il y a au plus $p * b$ noeuds en mémoire en même temps) mais une complexité temporelle de $O(b^p)$ (nous avons besoin de visiter tous les noeuds de l'arbre).

Essayons maintenant d'appliquer cet algorithme pour déterminer les valeurs minimax pour le jeu dans Figure 4. Nous appelons MINIMAX(a). Comme a n'a pas encore de valeur, nous mettons $v = -\infty$ (c'est un noeud `max`), puis nous commençons la boucle pour. Nous appellons donc MINIMAX(b), initialisant v à $+\infty$, et appellant MINIMAX sur ses trois fils e, f, et g. Comme e est un noeud terminal, nous renvoyons sa valeur d'utilité (0) et nous mettons v = MIN(MINIMAX(e),v) = 0. Pour f, nous renvoyons sa valeur, et v reste inchangé car MIN$(0, 15) = 0$. Même chose pour g, car sa valeur (c.à.d 5) est supérieure à 0. Nous avons terminé la boucle "pour" ; nous renvoyons donc 0 comme valeur minimax de b, et nous mettons $v = 0$ pour a. Puis nous appelons MINIMAX sur le deuxième fils c de a, et puis sur les trois petits-enfants

terminaux h, i, et j. Nous renvoyons les valeurs d'utilité pour ces trois, et nous déterminons la valeur de c. Elle est de -23 parce que MIN(MIN(MIN($+\infty$, -1), -23), 7)= -23. La valeur de a ne bouge pas comme -23 est plus petit que la valeur actuelle 0. Nous continuons avec le calcul avec le troisième fils de a. Nous appelons MINIMAX sur d, puis sur les fils k, l, et m qui renvoient leurs valeurs d'utilité. L'appel sur d retourne -2 car c'est le minimum des valeurs de k, l, et m. Puis nous revenons au calcul de la valeur de a. Comme 0 est le maximum de 0 et -2 et que nous n'avons plus de fils à considérer, nous retournons 0. Nous pouvons conclure que le joueur `max` ne peut pas gagner contre un adversaire qui joue de façon optimale, mais il peut garantir un match nul en choisissant b pour son premier coup.

L'algorithme minimax peut être facilement étendu au cas où il y a plus de 2 joueurs (dans ce cas, il nous faut un vecteur d'utilités pour en prendre compte les utilités des différents joueurs). Il est aussi possible de l'étendre aux jeux stochastiques, en ajoutant les noeuds supplémentaires aux arbres de jeu pour prendre en compte l'aspect stochastique (e.g. des noeuds pour réprésenter les lancers de dés) et en utilisant les probabilités des différentes configurations résultantes dans le calcul des valeurs minimax. Cette variation de l'algorithme classique est connue sous le nom *expectiminimax*.

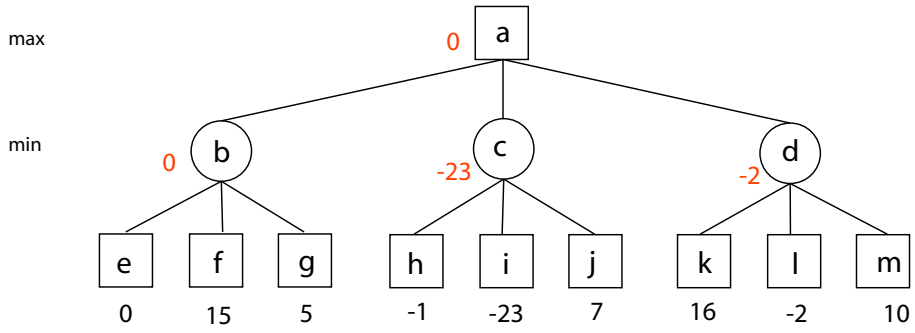

FIGURE 5 – L'algorithme minimax sur notre deuxième exemple.

Algorithme Alpha-Beta

L'algorithme minimax visite chaque noeud de l'arbre de jeu, mais est-ce vraiment nécessaire ? Dans le pire des cas, un parcours complet ne peut pas être évité, mais dans beaucoup d'autres cas, nous n'avons pas besoin

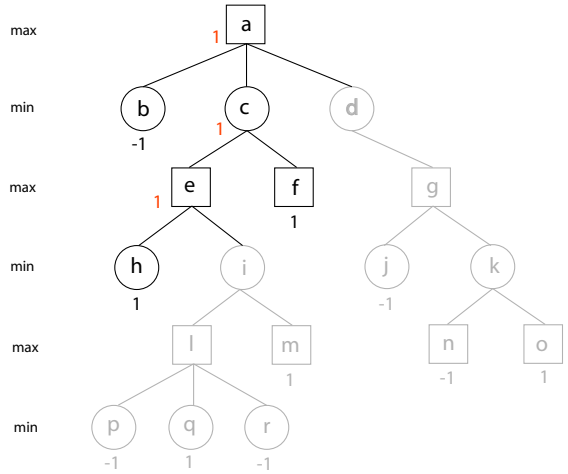

FIGURE 6 – Le calcul de la valeur minimax de a. Les parties grises de l'arbre ne sont pas examinées lors de la recherche.

de tout examiner pour déterminer le meilleur coup. Prenons par exemple l'arbre de jeu dans Figure 2. Vous avez peut-être remarqué qu'après avoir visité h, nous savions déjà que `max` peut gagner depuis l'état e. Nous aurions pu donc donner tout de suite la valeur de 1 à e *sans regarder le noeud* i *ni ses descendants*. Puis quand nous avions déterminé la valeur de c(qui est de 1), nous n'avions plus besoin de considérer le noeud d ni ses successeurs parce que nous avions déjà trouvé une stratégie gagnante pour `max`. Dans Figure 6, nous montrons la partie de l'arbre qui serait examinée si nous faisions le calcul des valeurs d'une manière intelligente. Comme vous voyez, avec un peu de raisonnement, nous avons réussi à éliminer significativement le nombre de noeuds examinés.

Ces observations sont à la base de *l'algorithme alpha-beta*, qui est une version plus sophistiquée de l'algorithme minimax. L'algorithme alpha-beta calcule la valeur minimax d'un noeud sans nécessairement calculer les valeurs de tous ses descendants. Le nom de cet algorithme vient des deux variables α et β qui sont utilisées pour stocker les valeurs des meilleurs options pour les joueurs `max` et `min` respéctivement. Si nous arrivons à un noeud `min` n qui a un fils dont la valeur est pire (ou égale) à la valeur α (meilleure possibilité actuelle pour le joueur `max`), nous savons que `max` ne va jamais choisir une action qui amène à n : il peut obtenir un meilleur résultat (au moins α) en choissisant un autre coup. Inversement, si nous sommes à un noeud

`max` n qui a un fils dont la valeur est supérieure à la valeur β (qui donne la meilleure possibilité examinée pour `min`),il est inutile de continuer à explorer cette branche car `min` ne choisira de toute façon pas n. Voici l'algorithme détaillé :

fonction VALEUR-ALPHA-BETA(n)
  retourner ALPHA-BETA(n, v_{min}, v_{max}) [2]
fin

fonction ALPHA-BETA(n, α, β)
  si n est un noeud terminal **alors**
    retourner la valeur d'utilité de n
  sinon
    si n est un nœud de type `min` **alors**
      v = $+\infty$
      pour tout fils f de n **faire**
        v = MIN(v, ALPHA-BETA(f, α, β))
        si $v \leq \alpha$ **alors** /* arrèter la recherche */
          retourner v
        β = MIN(β, v)
      fin pour
      retourner v
    sinon
      v = $-\infty$
      pour tout fils f de n **faire**
        v = MAX(v, ALPHA-BETA(f, α, β))
        si $v \geq \beta$ **alors** /* arrèter la recherche */
          retourner v
        α = MAX(α, v)
       fin pour
      retourner v
    fin si
  fin si
fin

Cet algorithme est très proche de l'algorithme minimax à ceci près que maintenant nous gardons en mémoire les valeurs des meilleures possibilités

2. v_{min} et v_{max} sont le maximum et minimum des valeurs d'utilités des noeuds terminaux. Il est courant dans la littérature d'utiliser systématiquement $-\infty$ et $+\infty$ comme valeurs initiales pour α et β. Quand $v_{max} = +\infty$ et $v_{min} = -\infty$, les deux alternatives donnent le même résultat, mais quand $v_{max} < +\infty$ ou $v_{min} > -\infty$, l'utilisation de v_{max} et v_{min} permet souvent une réduction plus importante de l'espace de recherche.

pour `min` et `max` (en utilisant les variables α et β). Quand nous arrivons à une configuration qui ne sera quoi qu'il arrive jamais choisie par un joueur (car il y a mieux ailleurs) nous arrêtons la recherche sur cette partie de l'arbre. Il est important de remarquer que les valeurs retournées par alpha-beta (et donc les choix de coups) sont exactement les mêmes qu'avec l'algorithme minimax. Donc nous pouvons utiliser alpha-beta sans perdre la garantie de l'optimalité.

Essayons maintenant d'appliquer l'algorithme alpha-beta à l'arbre de jeu de la Figure 4. Nous appellons la fonction VALEUR-ALPHA-BETA sur la racine a, ce qui lance un appel à ALPHA-BETA avec comme arguments le noeud a, la valeur $\alpha = -23$, et la valeur $\beta = 16$. Nous appellons ensuite l'algorithme sur le noeud b (gardant $\alpha = -23$ $\beta = 16$), et puis sur son fils e. Ce dernier est un noeud terminal, donc nous renvoyons sa valeur d'utilité (c.à.d. 0). Nous avons maintenant $v = 0$ et $\beta = 0$ (nous sommes toujours dans l'appel de b). Puis nous appelons l'algorithme sur f et g, mais la valeur v de b ne change pas car les valeurs d'utilité de f et g sont supérieures à 0. Nous renvoyons donc la valeur 0 pour b. Ceci nous permet de mettre à jour la valeur α pour a parce que maintenant nous savons que nous ne pouvons pas faire pire que 0. Nous continuons avec un appel pour c (maintenant avec $\alpha = 0$ et $\beta = 16$). Nous faisons un autre appel pour son fils h qui nous renvoie sa valeur d'utilité -1. Nous mettons donc $v = -1$, et nous renvoyons cette valeur pour c car nous avons $-1 = v \leq \alpha = 0$. Nous revenons donc à l'appel initial de a, et nous appellons ensuite la fonction sur le dernier fils d de a. Nous cherchons d'abord la valeur du petit-fils k, et nous trouvons $v = 16$. Puis nous passons au deuxième fils l dont la valeur est égale à -2. Nous renvoyons cette valeur pour d sans regarder le dernier fils m car nous avons $-2 = v \leq \alpha = 0$. Nous avons donc déterminé la valeur de a (qui est 0), ce qui est exactement ce que nous avions trouvé avec l'algorithme minimax sauf que cette fois nous n'avons pas visité tous les noeuds de l'arbre.

L'algorithme alpha-beta peut réduire significativement le nombre de noeuds examiné pour calculer la stratégie optimale, mais son efficacité est très dépendante de l'ordre dans lequel les noeuds de l'arbre sont examinés. Si nous suivons le pire ordre, nous ne réalisons pas de gain de temps parce que nous explorons tout l'arbre, comme avec minimax. Si par contre nous suivons l'ordre optimal, l'algorithme alpha-beta donne une complexité temporelle de $\mathcal{O}(b^{p/2})$ au lieu de $\mathcal{O}(b^p)$ pour l'algorithme classique minimax. La complexité reste donc exponentielle, mais nous pouvons maintenant examiner un arbre avec un profondeur deux fois plus grande, ce qui n'est pas du tout négligeable. Bien sûr, nous ne savons pas déterminer l'ordre idéalle, mais en pratique il est souvent possible de trouver des heuristiques qui permettent à approcher de très près cette limite théorique.

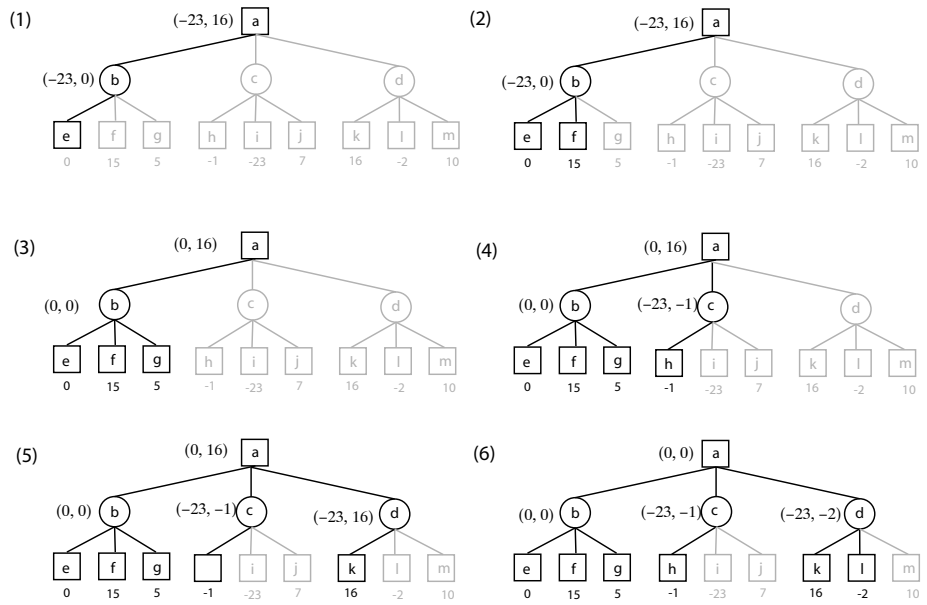

FIGURE 7 – L'algorithme alpha-beta sur notre deuxième exemple. Les valeurs minimax possibles pour chaque noeud aux différentes stages du calcul sont notées. Rémarquez que nous ne déterminons pas la valeur précise des noeuds c et d car elles ne sont pas nécéssaires pour établir la valeur de la racine a. Notez également que les noeuds i, j, et m ne sont pas examiné lors de la recherche.

Notons que l'algorithme alpha-beta ne se généralise pas très bien aux jeux à plus de deux joueurs. Il peut être utilisé avec expectiminimax pour des jeux stochastiques, mais le gain de temps est en général bien moins importants que pour les jeux déterministes.

Décisions en Temps Réel

L'algorithme alpha-beta est une amélioration importante de l'algorithme minimax, mais nous avons toujours besoin de considérer un nombre exponentiel de noeuds, ce qui le rend inutilisable en pratique car nous n'avons qu'un temps limité pour prendre les décisions. Dans cette section, nous détaillons plusieurs techniques qui peuvent être utilisées pour rendre alpha-beta utilisable en pratique.

Fonction d'évaluation Il est généralement impossible faute de temps de rechercher jusqu'aux noeuds terminaux. Pour pallier ce problème, nous introduisons une *fonction d'évaluation* qui donne une estimation de la valeur minimax d'un noeud non-terminal. Dans cette version modifiée de alpha-beta, nous ajoutons un test pour savoir à quel moment nous devrions arrèter la recherche. Quand nous nous arrètons sur un noeud non-terminal, nous utilisons la fonction d'évalution pour donner une valeur à ce noeud (pour les noeuds terminaux, nous utilisons leurs valeurs d'utilité, comme d'habitude). L'utilisation d'une fonction d'évaluation est la modification la plus importante pour permettre la prise de décisions en temps réel. L'éfficacité de l'algorithme dépendra bien sûr de la précision de la fonction d'évaluation. Les fonctions d'évaluation sont souvent codé à la main avec l'aide des experts humains et des résultats de nombreux experiences, mais elles peuvent aussi être généré via l'apprentissage automatique. C'est notamment le cas pour le programme TD-GAMMON (un programme pour le backgammon) qui a appris une fonction d'évalution en utilisant des réseaux de neurones. TD-GAMMON a amélioré la précision de sa fonction en jouant contre elle-même, et est aujourd'hui parmi les meilleurs joueurs du monde.

Approfondissement séléctive La façon la plus facile d'implémenter la version modifiée de alpha-beta que nous venons de voir serait de limiter le profondeur de la recherche. Quand nous atteindrons cette profondeur, nous arrètons la recherche et renvoyons la valeur donné par la fonction d'évaluation. Cette approche a l'avantage d'être simple (nous ne perdons pas de temps à décider si nous continuons ou pas), mais n'est pas toujours le plus approprié parce qu'elle traite tous les noeuds (et coups)

à égalité. Il est souvent préférable de dédier plus de temps à certaines parties de l'arbre que d'autres. Par exemple, nous pourrions donner la priorité aux coups qui semblent très prometteux pour les examiner plus en détaille. Cette technique, qui est appellé *"singular-extension search"* en anglais, est une des components clés de la programme Deep Blue dont la recherche peut parfois atteindre une profondeur de 80 coups pour des combinaisons de coups suffisament intéréssants (au lieu des 28 coups examiné normalement). Une autre technique qui est utilisé par Deep Blue et par Chinook (la meilleure programme pour les dames) est la recherche de*quiescence* qui donne plus de temps aux séquences de coups où la valeur estimée est très variable (car ceci veut dire que la fonction d'évaluation est peu fiable pour ces configurations et pourrait bénéficier de plus d'examen).

Heuristiques pour ordonner la recherche Nous avons mentionné dans la dernière section l'importance qui joue l'ordre dans laquelle les noeuds sont examinés, car une bonne ordre peut conduire à une réduction substantielle de l'éspace de recherche. Il est donc intéréssant d'essayer d'utiliser les heuristiques pour mieux ordonner la recherche. Une possibilité est d'utiliser la fonction d'évaluation comme heuristique pour choisir l'ordre des noeuds. Une autre possibilité est de garder en mémoire des "bons coups" qui ont permis une réduction considérable du nombre de noeuds dans une autre partie de l'arbre et d'essayer ces coups en priorité. L'heuristique de l'historique, comme elle est appellé, s'avère très performante, et elle joue un role important dans le succès de Chinook.

Table de transpositions Dans le chapitre sur la recherche, nous avons parlé du problème qui arrivent quand l'arbre contient plusieurs fois le même état. Ce même problème arrive dans les arbres de jeu car il y a souvent plusieurs suites de coups qui ammènent à la même configuration. Comme il est inefficace de recalculer à plusieurs reprises la valeur d'une configuration, il est souvent intéréssant de garder en mémoire les valeurs déjà calculé. La table contenant les configurations et leur valeurs est connu sous le nom d'une table de transpositions. Bien entendu, nous aurions pas suffisamment de mémoire de tout sauvegarder, donc nous aurions besoin de faire une choix parmi les configurations à stocker. La bonne utilisation d'une table de transpositions peut avoir des effets très importants sur l'éfficacité de la recherche. Par exemple, en échècs, il est parfois possible d'examiner les noeuds d'une profondeur deux fois plus grandes grace à cette technique. Les tables de transpositions sont exploités par Deep Blue et Chinook.

Mémorisation L'idée est de garder en mémoire des informations qui peuvent

être utiles pour séléctionner parmi des coups ou pour améliorer l'éfficacité de la recherche. Par exemple, nous pourrions mémoriser un ensemble de bonnes séquences de coups initiales qui pourraient guider le choix des coups au début de jeu (où nous sommes le plus vulnerable car nous sommes plus loin des configurations terminales). Une autre possibilité serait de stocker les valeurs des configurations qui sont à quelques coups près d'une configuration terminale. Cette technique permet de réduire le nombre de coups nécéssaire pour arriver à une configuration avec valeur connue. La fameuse programme d'échècs Deep Blue compte quelques 4000 combinaisons initiales, 700,000 jeux entiers des grandmaîtres, et les valeurs minimax pour toutes les configurations composé d'au plus cinq pièces. Le programme Chinook (qui est de loin le plus perfomant pour les dames) comprend les valeurs de toutes les configurations d'au plus huit pièces, ainsi qu'une bibliothèque de 60,000 ouvertures.

En conjonction avec les techniques ci-dessous, l'algorithme alpha-beta peut être très performant. "indeed", les meilleures programmes pour les échècs, les dames, et l'othello (parmi d'autres) utilisent toutes l'algorithme alpha-beta en combinaison avec les techniques que nous venons de présenter.

Chapitre 5: Représentation des Connaissances et Raisonnement Automatique

Si un agent veut agir de façon "intelligente" dans un environnement donné, il est important que cet agent ait des connaissances relatives à cet environnement. Pour concevoir un agent doué d'une "intelligence artificielle", deux problèmes majeurs se posent à nous : comment, d'une part, représenter les connaissances de l'agent (par exemple : "il fait nuit"), et comment d'autre part raisonner à partir de ces connaissances (par exemple : "puisqu'il fait nuit, les magasins sont fermés") ? Ces deux problématiques sont interconnectées car la façon dont on représente les connaissances a un impact important sur le type de raisonnements qui peuvent être effectués ainsi que sur la complexité (calculatoire) du raisonnement.

1 Représentation des connaissances

La représentation des connaissances, comme son nom le suggère, a pour but l'étude des formalismes qui permettent la représentation de toutes formes de connaissances. En général, ce sont des langages logiques ou probabilistes. Dans ce chapitre, nous allons nous concentrer sur les langages logiques. Nous commençons donc par un petit rappel des bases de la logique propositionnelle et la logique du premier ordre, puis nous considérons quelques autres logiques.

Rappel : Logique propositionnelle

La logique propositionnelle est une logique très simple qui se trouve à la base de presque toutes les logiques qui sont étudiées aujourd'hui. Les éléments de bases sont des *propositions* (ou variables propositionnelles) qui représentent des énoncés qui peuvent être soit vrais soit faux dans une situation donnée. Nous pourrions par exemple avoir une proposition p qui représente "Marc est un étudiant" ou une proposition q qui représente l'énoncé "Deux est un nombre pair". Des formules complexes peuvent être construites à partir des propositions en utilisants les connectives logiques : $\wedge$ ("et"),

$\vee$ ("ou"), et $\neg$ (negation). Par exemple, nous pourrions avoir les formules $p \wedge q$ ("Marc est un étudiant *et* deux est un nombre pair"), $p \vee q$ ("Marc est un étudiant *ou* deux est un nombre pair"), et $\neg p$ ("Marc *n'*est *pas* un étudiant"). Souvent pour simplicité nous introduisons aussi les connecteurs $\rightarrow$ et $\leftrightarrow$. $p \rightarrow q$ est une abbréviation pour $\neg p \vee q$, et $p \leftrightarrow q$ est juste une façon plus concis d'écrire $(p \wedge q) \vee (\neg p \wedge \neg q)$.

Dans la deuxième partie du chapitre, nous allons avoir besoin des notions de littéral et de clause. Je vous rappelle qu'un *littéral* est une proposition ou la négation d'une proposition. Par exemple, p et $\neg q$ sont des littéraux. Une *clause* est une disjonction de littéraux. Des formules p et $p \vee \neg q \vee r$ sont des exemples de clauses.

Un *modèle* M est une fonction qui donne une valeur de vérité (soit $vrai$, soit $faux$) à chaque proposition du langage. Une proposition p est satisfaite dans un modèle M si elle réçoit la valeur $vrai$, et elle n'est pas satisfaite si elle reçoit la valeur $faux$. Dans le premier cas, nous écrivons $M \models p$ (M est un modèle de p), et dans le deuxième cas, nous écrivons $M \not\models p$ (M n'est pas un modèle de p). La *satisfaction* des formules dans des modèles est entièrement détérminée par les valeurs de vérité des propositions de la formule :

- $M \models \phi \wedge \psi$ si et seulement si $M \models \phi$ et $M \models \psi$
- $M \models \phi \vee \psi$ si et seulement si $M \models \phi$ ou $M \models \psi$ (ou les deux)
- $M \models \neg\phi$ si et seulement si $M \not\models \phi$

Une formule est *valide* si elle est satisfaite par tout modèle, et elle est dite *satisfiable* si elle est satisfaite dans au moins un modèle. Une formule ϕ est une *conséquence logique* de ψ (ce que l'on note $\phi \models \psi$) si chaque modèle qui satisfait ϕ satisfait aussi ψ, ou autrement dit, s'il n'y a pas de modèle de $\phi \wedge \neg\psi$.

La logique propositionnelle est *decidable*, c'est-à-dire qu'il existe des algorithmes pour déterminer si une formule est valide ou non. Le problème de validité est co-NP-complet, et le problème dual de satisfiabilité est NP-complet. Ces résultats de complexité montrent que nous ne pouvons pas ésperer de trouver des algorithme de validité ou de satisfiabilité qui terminent dans un temps raisonnable sur toutes les formules [1]. C'est plutôt une mauvaise nouvelle, mais il faut se rappeller aussi qu'il s'agit de la complexité dans le pire cas. En pratique, la recherche actuelle fournit des algorithmes de plus en plus performants (grace aux heuristiques) et peuvent aujourd'hui traiter des formules de taille importante qui n'auraient pas été résolubles il y a quelques années.

1. Sauf peut-être dans le cas où P=NP, ce qui est considéré comme très improbable.

Rappel : Logique du premier ordre

La logique du premier ordre est une logique très expressive et bien étudiée. Les formules en logique du premier ordre sont formée des éléments suivants :

- les symboles pour constants, e.g. $Marie$, $gareMarseille$
- les variables, e.g. x, y
- les symboles pour fonctions, e.g. $méreDe$
- les symboles pour prédicats, e.g. $humain$, $dans$
- les connectives logiques : $\wedge$, $\neg$, etc.
- la quantificateur universelle $\forall$
- la quantificateur existentielle $\exists$
- la symbole d'égalite $=$

Notez que chaque symbole pour fonction et chaque symbole pour prédicat a un *arité* qui détérmine combien d'arguments elle peut prendre (0, 1, 2, ...), e.g. $mèreDe$ a un arité de 1, $dans$ a un arité de 2. Les *termes* sont des symboles pour constantes, des symboles pour variables, ou bien des symboles pour fonctions appliquées à d'autres termes, e.g. $Marie$, $méreDe(x)$, $méreDe(méreDe(Marie))$. Les *atomes* sont de deux types : soit $P(terme_1, ..., terme_n)$ où P est un symbole pour prédicat avec arité n et les $terme_i$ sont des termes, soit $terme_1 = terme_2$ où $terme_1$ et $terme_2$ sont des termes. Nous pourrions par exemple avoir des atomes $humain(Marie)$, $humain(x)$, $dans(Marie, gareMarseille)$ ou bien $Marie = méreDe(x)$. Les formules sont soit des atomes, e.g. $humain(Marie)$, soit des combinaisons booléenes de formules, e.g. $dans(Marie, gareMarseille) \wedge humain(Marie)$, soit les formules préfixées par $\forall x$ ou $\exists x$ (où x est un variable), e.g. $\exists x.humain(x)$ ou $\forall x.humain(x)$.

Un modèle en logique du premier ordre est composé d'un ensemble non-vide U (appelé l'univers) et d'une fonction d'interpretation I. La fonction I associe à chaque symbole pour constante C un élément de l'univers $I(C) \in U$, à chaque symbole pour fonction F (où l'arité de F est n) une fonction n-aire de U à U, et à chaque symbole pour prédicat P (où l'arité de P est n) un prédicat n-aire sur U. Pour traiter des variables (qui ne sont pas fixés par des modèles), nous faisons appel aux valuations, qui sont des fonctions qui associent à chaque variable un membre de l'univers U. Dans la définition de la satisfaction, nous aurions aussi besoin de la notation $v(x/d)$, qui donne la valuation qui est la même que v sauf pour la variable x, à laquelle on associe l'élément de l'univers d. La satisfaction d'une formule à par rapport un modèle $M =< U, I >$ et une valuation v comme ceci [2] :

- $\models_{M,v} t_1 = t_2$ ssi t_1 et t_2 référe au même élément de U
- $\models_{M,v} P(t_1, ..., t_n)$ ssi le tuple d'éléments associés à $(t_1, ..., t_n)$ est dans $I(P)$

2. Pour gagner de la place, j'utilise "ssi" comme abbréviation pour "si et seulement si".

– $\models_{M,v} \phi \wedge \psi$ ssi $\models_{M,v} \phi$ et $\models_{M,v} \psi$
– $\models_{M,v} \phi \vee \psi$ ssi $\models_{M,v} \phi$ ou $\models_{M,v} \psi$
– $\models_{M,v} \neg\phi$ ssi $\not\models_{M,v} \phi$
– $\models_{M,v} \forall x.\phi$ ssi pour tout $d \in U$ nous avons $\models_{M,v(x/d)} \phi$
– $\models_{M,v} \exists x.\phi$ ssi il existe $d \in U$ tel que $\models_{M,v(x/d)} \phi$

Une formule est valide si elle est satisfaite par tout modèle et toute valuation.

La logique du premier ordre est beaucoup plus puissante que la logique propositionnelle, et cette expressivité accrue se paie par une difficulté du calcul. En effet, à l'inverse de la logique propositionnelle, la logique du premier ordre n'est pas decidable : il n'existe pas d'algorithmes qui retournent oui si une formule est valide, et non si elle ne l'est pas.

D'autres logiques

La logique propositionnelle et la logique du premier ordre sont de loin les deux logiques les plus étudiées, mais on constate un intérêt grandissant pour d'autres logiques qui sont parfois plus adaptées pour représenter certains types de connaissances. En voici quelques-unes :

Logiques de description Cette famille de logiques se trouve à mi-chemin entre la logique propositionnelle et la logique du premier ordre, offrant ainsi une expressivité beaucoup plus importante que celle de la logique propositionnelle mais avec une complexité du raisonnement moindre que pour la logique du premier ordre (en particulier, les logiques de description sont souvent decidables). Ces logiques possèdent deux types de formules : des axiomes qui décrivent des relations entre des concepts et des assertions qui expriment les caractéristiques des individus et les relations entre individus. Par exemple, nous pourrions avoir des axiomes $Oiseau \sqsubseteq Animal$ (= chaque oiseau est un animal) et $Oiseau \sqsubseteq \forall enfant.Oiseau$ (= les enfants des oiseaux sont aussi des oiseaux) et des assertions $Oiseau(Tweety)$ (=Tweety est un oiseau) et $enfant(Tweety, Paul)$ (=Paul est un enfant de Tweety). Nous pourrions alors inférer que Paul est un oiseau et que Tweety et Paul sont tous les deux des animaux. Les logiques de description sont utilisées dans plusieurs domaines d'application (e.g. la médecine, le traitement du langage naturel, etc.) et sont à la base de la web sémantique (qui est censé être la prochaine incarnation du Web).

Logiques temporelles Ces logiques sont obtenues à partir de la logique propositionnelle en ajoutant un certain nombre de quantificateurs temporels, e.g. "maintenant", "dorénavant", "toujours", qui permettent de

parler des propriétés qui sont vraies à différents moments dans le temps. On pourrait par exemple exprimer dans une telle logique le fait "Marc est un étudiant maintenant mais un jour il ne sera plus étudiant" ou "Deux est toujours un nombre pair". L'une des applications principales des logiques temporelles est la vérification de programmes où ces logiques sont utilisées pour décrire des propriétés qui devraient être vraies lors d'une exécution d'un programme.

Logique floue Dans les logiques classiques, les propositions sont soit vraies soit fausses. Néanmoins, il existe des propriétés qui semblent pouvoir être vraies à un certaine degré. Prenons par exemple la propriété "grand". Il serait un peu bizarre de séparer les êtres humains entre deux classes, ceux qui sont grands, et ceux qui ne le sont pas – comment chosir une hauteur telle que toutes les personnes avec une taille supérieur à cette hauteur sont grands, et les autres pas ? La logique floue tente de répondre à ce problème en permettant aux propositions de prendre les valeurs entre 0 et 1, où 0 signifie que la proposition n'est pas satisfaite du tout, et 1 signifie la satisfaction complète de la proposition, et les valeurs entre 0 et 1 signifie des niveaux de satisfaction de la proposition. Nous pourrions donc dire que quelqu'un est grand à niveau 0.3 (ce qui veut dire que la personne n'est pas tellement grande mais pas complètement petite) ou à niveau 0.8 (ce qui dit que la personne est plutôt grande). La logique floue a été appliquée avec succès à de nombreux problèmes dans l'industrie.

Logiques de connaissances et/ou croyances L'une des caractéristiques du raisonnement humain est sa faculté de raisonner sur ses propres connaissances/croyances et sur les connaissances/croyances des autres. Un certain nombre de logiques ont été proposées pour formaliser ce type de raisonnement. Dans ces logiques, nous pouvons exprimer les informations de type "je sais que Marie a un enfant, et je crois que c'est une fille, mais je ne suis pas sûr" ou bien "je crois qu'il croit que je sais qui a volé le diamant, mais je ne le sais pas". Le raisonnement sur les connaissances devient un sujet de plus en plus populaire grâce à son utilité dans de nombreux domaines tels que l'IA, la linguistique, l'informatique distribuée, et l'économie.

Logiques non-monotones Les logiques classiques sont appelées *monotones* parce que si $\phi \models \psi$ il est forcément le cas que $\phi \wedge \zeta \models \psi$. Autrement dit, le fait d'ajouter de l'information supplémentaire ne peut pas causer de perte d'informations. Or cette propriété ne semblent pas toujours vérifiée dans les raisonnements humains. Pour prendre un exemple classique, supposons que je vous dit que Tweety est un oiseau, et je vous

demande si Tweety vole. Vous allez probablement me dire que oui car les oiseaux volent. Mais si maintenant j'ajoute le fait que Tweety est un pingouin, vous allez maintenant me dire que Tweety ne vole pas. C'est un exemple d'un raisonnement non-monotone car l'ajout d'une hypothèse (Tweety est un pingouin) fait perdre l'une des conséquences (Tweety vole). Ce type de raisonnement ne satisfait pas donc les lois de la logique classique mais il est rationnel tout de même car il nous permet de raisonner en l'absence d'informations complètes. Les logiques non-monotones essaient de formaliser ce type de raisonnement. Dans la logique des défauts, par exemple, nous pourrions traiter l'exemple de Tweety en utilisant une règle de type "Si X est un oiseau *et* s'il n'est pas connu que X est un pingouin *alors* conclure que X vole". La conclusion que Tweety vole reste valide tant qu'on ne sait pas que Tweety est un pingouin.

Raisonnement automatique

Dans cette section, nous considérons les différents types du raisonnement, puis nous introduisons une méthode du raisonnement pour la logique propositionnelle.

Types de raisonnement

On peut distinguer trois types de raisonnement :

Raisonnement déductif : C'est le type de raisonnement le plus étudié. Il s'agit de dériver les conséquences d'un ensemble d'informations données. Le raisonnement déductif permet de déduire "Marie est un docteur" à partir des informations "Marie est un docteur ou un professeur" et "Marie n'est pas un professeur", ou bien que "Tweety a des ailes" à partir de "Tweety est un oiseau" et "Tout oiseau a des ailes". L'une des caractéristiques du raisonnement déductif est que si nous dérivons une conséquence d'un ensemble d'informations qui sont vraies, alors nous sommes sûrs que la conséquence est vraie aussi.

Raisonnement abductif : Le but de l'abduction est d'expliquer des observations. Le raisonnement abductif est utilisé couramment dans la vie de tous les jours, et notamment dans le diagnostic médical : nous cherchons une cause possible (maladie) qui pourrait expliquer les observations (des symptômes de la patiente) en présence de connaissances générales (les connaissances sur la médécine). Un exemple d'un raisonnement abductif pourrait donc être "La patiente a la grippe" étant

donné que "La patiente a de la fièvre, de la toux, et mal à la tête" et "La grippe peut provoquer des fièvres, maux de gorge, maux de tête, fatigue, toux, et des douleurs musculaires". Notez qu'en général il y a plusieurs explications possibles, e.g. les symptômes de la patiente pourrait aussi être provoqué par une bronchite combinée avec un cancer du cerveau. En général, nous préférons les explications les plus simples, e.g. moins de maladies, ou des plus probables, e.g. des maladies courantes.

Raisonnement inductif : Il s'agit ici de produire des connaissances générales à partir d'observations. Le raisonnement inductif peut donc être vu comme une forme d'apprentissage. Un exemple d'un raisonnement inductif serait de conclure que "Tous les cygnes sont blancs" étant donné que tous les cygnes que l'agent a vu dans sa vie étaient blancs. Il est important de noter qu'à la différence du raisonnement déductif, les conclusions du raisonnement inductif peuvent s'avérer fausses (il existe des cygnes noirs !). Néanmoins, ce type de raisonnement est très important car il nous permet d'agir dans un environnement donné, lorsque les informations dont on dispose sur celui-ci sont incomplètes.

Dans ce chapitre, nous allons nous focaliser sur le raisonnement déductif en logique propositionnelle. Le raisonnement inductif sera abordé dans le prochain chapitre.

Raisonnement déductif en logique propositionnelle

Il y a deux problèmes principaux lié au raisonnement déductif, le premier étant de tester si une formule est bien une conséquence logique d'une autre formule, et le deuxième étant de produire les conséquences d'une formule donnée.

Vous avez peut-être déjà vu des méthodes pour résoudre le premier de ces problèmes. La plus facile à comprendre (mais aussi la moins efficace) est la méthode de table de vérite. Une formule ψ est une conséquence logique d'une formule ϕ juste dans le cas où chaque modèle de ϕ est aussi un modèle de ψ. Nous pourrions donc énumérer tous les modèles possibles (c'est le principe même de la table de vérité) et vérifier que cette condition est vraie pour chacun des modèles. Une autre façon de résoudre ce problème est de le transformer en problème de satisfiabilité : une formule ψ est une conséquence logique d'une formule ϕ si et seulement si la formule $\phi \wedge \neg\psi$ n'est pas satisfiable. Nous pouvons donc utiliser des algorithmes de satisfiabilité (par exemple, l'algorithme DPLL) pour tester si une formule est une conséquence d'une autre.

Ces algorithmes ne donnent pas, par contre, une façon de dériver les

conséquences d'une formule. C'est la raison pour laquelle nous introduisons une autre méthode, appelée la règle de résolution, qui sert à la fois à identifier les conséquences d'une formule et aussi à les générer. Une autre raison d'étudier cette méthode est qu'elle peut être généralisée à la logique du premier ordre. En effet, une bonne partie des algorithmes pour la logique du premier ordre sont basés sur la résolution.

La règle de résolution

Supposons que je vous dise que s'il fait beau, je vais à la plage, et s'il ne fait pas beau, je vais au cinéma ou je reste à travailler à la maison. Vous allez pouvoir conclure que je vais à la plage ou je vais au cinéma ou je reste à la maison à travailler. En logique, nous pourrions formaliser cet argument ainsi (rappelons que $a \rightarrow b \equiv \neg a \vee b$) :

$$\frac{\neg beau \vee plage \qquad beau \vee cinéma \vee travail}{plage \vee cinéma \vee travail}$$

La règle de résolution est juste une généralisation de ce type d'argument. Si nous avons deux clauses, et il y a une proposition l^* dans l'une des clauses dont la négation $\neg l^*$ se trouve dans l'autre clause, nous pouvons appliquer la règle de résolution pour obténir une nouvelle clause qui est la disjonction des deux clauses d'origine mais avec les littéraux l^* et $\neg l^*$ supprimés. Plus formellement, la règle de résolution peut être écrite ainsi :

$$\frac{l_1 \vee ... \vee l_j \vee ... \vee l_n \qquad l'_1 \vee ... \vee l'_k \vee ... \vee l'_m}{l_1 \vee ... \vee l_{j-1} \vee l_{j+1} \vee ... \vee l_n \vee l'_1 \vee ... \vee l'_{k-1} \vee l'_{k+1} \vee ... \vee l'_m}$$

où l_j et l'_k sont des littéraux opposés. Notez qu'ici nous avons supposé que chaque clause contient au plus une seule copie de chaque littéral. Si ce n'est pas le cas, il faut supprimer toutes les copies du littéral dans la clause. Par exemple, le résultat de la résolution de $a \vee a \vee b$ avec $\neg a \vee c$ est $b \vee c$ et non pas $a \vee b \vee c$.

Nous allons voir dans un moment comment la règle de résolution peut être utilisée pour tester la satisfiabilité d'une formule (et donc pour tester si une formule est une conséquence logique d'une autre formule) et aussi pour générer des conséquences d'une formule. Mais tout d'abord, il nous faut comprendre comment mettre les formules dans la bonne forme syntactique pour l'application de la règle de résolution.

Mise sous forme clausale

Pour pouvoir appliquer la règle de résolution à une formule, il faut que la formule soit sous forme clausale. Ce n'est pas un problème parce que toute

formule propositionnelle est équivalente à une conjonction de clauses. Pour transformer une formule en forme clausale, il suffit d'appliquer les règles suivantes :

1. Si les abbréviations $\rightarrow$ et $\leftrightarrow$ sont présentes dans la formule, supprimez-les en utilisant les équivalences $\phi \rightarrow \psi \equiv \neg\phi \vee \psi$ et $\phi \leftrightarrow \psi \equiv (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$.
2. Appliquez les équivalences $\neg(\phi \wedge \psi) \equiv \neg\phi \vee \neg\psi$ et $\neg(\phi \vee \psi) \equiv \neg\phi \wedge \neg\psi$ et $\neg\neg\phi \equiv \phi$ jusqu'à ce que les négations se trouvent juste devant les propositions.
3. Appliquez les équivalences $(\phi \wedge \psi) \vee \zeta \equiv (\phi \vee \zeta) \wedge (\psi \vee \zeta)$ pour arriver à une conjonction de clauses.

Nous illustrons la mise sous forme clausale de la formule $\neg(a \rightarrow b) \vee (c \wedge d)$:

$$\begin{aligned}
\neg(a \rightarrow b) \vee (c \wedge d) &\equiv \neg(\neg a \vee b) \vee (c \wedge d) \\
&\equiv (\neg\neg a \wedge \neg b) \vee (c \wedge d) \\
&\equiv (a \wedge \neg b) \vee (c \wedge d) \\
&\equiv (a \vee (c \wedge d)) \wedge (\neg b \vee (c \wedge d)) \\
&\equiv (a \vee c) \wedge (a \vee d) \wedge (\neg b \vee c) \wedge (\neg b \vee d)
\end{aligned}$$

La formule $\neg(\neg a \vee b) \vee (c \wedge d)$ est donc équivalente à l'ensemble de clauses $\{a \vee c, a \vee d, \neg b \vee c, \neg b \vee d\}$.

Vérification et génération de conséquences via la règle de résolution

Nous montrons maintenant comment utiliser la règle de résolution pour faire du raisonnement déductif.

Commençons par voir comment vérifier si une formule est une conséquence logique d'une autre formule. Pour cela, nous allons montrer comment tester la satisfiabilité d'une formule avec la règle de résolution. Comme nous avons déjà vu comment transformer le problème de tester si une formule est une conséquence logique à un problème de satisfiablilté, cela nous donne une méthode pour vérifier les conséquences.

Voici comment tester la satisfiabilité d'une formule ϕ avec la règle de résolution :

1. Transformer ϕ en un ensemble de clauses C.
2. Tant qu'il existe deux clauses cl_1 et cl_2 dans C telles que a) la règle de résolution s'applique à cl_1 et cl_2, et b) le résultat de la résolution de cl_1 avec cl_2 n'est pas déjà dans C et n'est pas une tautologie (=contient à

la fois un littéral est sa négation) : si le résultat de la résolution est la clause vide, retournez "pas satisfiable", sinon ajoutez le résultat de la résolution de cl_1 avec cl_2 à C.

3. Retournez "satisfiable".

Il peut être prouvé que cet algorithme s'arrête toujours et qu'il retourne "satisfiable" si et seulement si la formule est satisfiable.

Pour illustrer le fonctionnement de cet algorithme, nous l'utilisons pour tester si $\psi = cinéma$ est une conséquence logique de la formule suivante [3] :

$\phi = (beau \rightarrow plage) \wedge (\neg beau \rightarrow cinéma \vee travail) \wedge (paresseux \rightarrow \neg travail) \wedge \neg beau \wedge parresseux$

1. Nous transformons la formule $\phi \wedge \neg\psi$ en un ensemble de clauses :

   $C = \{\neg beau \vee plage, beau \vee cinéma \vee travail, \neg paresseux \vee \neg travail, \neg beau, paresseux, \neg cinéma\}$

2. Les clauses $\neg beau \vee plage$ et $beau \vee cinéma \vee travail$ satisfont les conditions, donc nous ajoutons la clause $plage \vee cinéma \vee travail$ à C.

3. Nous ajoutons la clause $cinema \vee travail$, qui est le résultat de la résolution de $beau \vee cinéma \vee travail$ et $\neg beau$, à l'ensemble C.

4. Nous appliquons la règle de résolution aux clauses $\neg paresseux \vee \neg travail$ et $parresseux$ et nous ajoutons la clause résultante $\neg travail$ à l'ensemble C.

5. Nous ajoutons $cinema$ à C après avoir fait la résolution de $cinema \vee travail$ et $\neg travail$.

6. L'algorithme retourne "pas satisfiable" car le résultat de la résolution de $cinema$ et $\neg cinema$ est la clause vide.

7. Nous pouvons retourner "oui" car le fait que $\phi \wedge \neg\psi$ n'est pas satisfiable implique que ψ est bien une conséquence de ϕ.

L'algorithme pour la génération de conséquence d'une formule ϕ est très similaire à celui que nous venons de voir :

1. Transformer ϕ en un ensemble de clauses C.

2. Tant qu'il existe deux clauses cl_1 et cl_2 dans C telles que a) la règle de résolution s'applique à cl_1 et cl_2, et b) le résultat de la résolution de cl_1 avec cl_2 n'est ni dans C ni une tautologie : ajoutez le résultat de la résolution de cl_1 avec cl_2 à C.

3. Souvent il y a plusieurs paires de clauses qui satisfont les conditions pour l'application de la règle de résolution. L'ordre dans lequel nous les traitons ne change pas le résultat de l'algorithme.

3. Enlever de C toutes les clauses qui ne sont pas minimales (une clause cl est *minimale* dans C s'il n'existe pas une autre clause cl' dans C telle que tous les littéraux de cl' sont dans cl mais certains des littéraux de cl ne sont pas dans cl').
4. Renvoyer C.

Cet algorithme termine toujours, et il est correct (chacune des clauses retournées est bien une conséquence de ϕ). L'algorithme n'est pas complet, au moins dans le sens habituel du terme, parce qu'il ne retourne pas toutes les conséquences de ϕ. Mais l'algorithme est complet dans un autre sens parce qu'il retourne toutes les conséquences clausales les plus fortes de ϕ. Ces clauses sont appellées les *impliqués premiers* de ϕ, et elles satisfont la propriété suivante :

– Si une clause cl est une conséquence d'une formule ϕ, il existe un impliqué premier p de ϕ tel que $p \models cl$.

Les impliqués premiers sont donc une façon compacte de représenter les conséquences d'une formule.

Essayez maintenant d'appliquer cet algorithme pour calculer les impliqués premiers de la formule suivante :

$\phi = (beau \rightarrow plage) \wedge (\neg beau \rightarrow cinéma \vee travail) \wedge (paresseux \rightarrow \neg travail) \wedge \neg beau \wedge parresseux$

1. Nous transformons la formule ϕ en un ensemble de clauses :

   $C = \{\neg beau \vee plage, beau \vee cinéma \vee travail, \neg paresseux \vee \neg travail, \neg beau, paresseux\}$
2. Les clauses $\neg beau \vee plage$ et $beau \vee cinéma \vee travail$ satisfont les conditions, donc nous ajoutons la clause $plage \vee cinéma \vee travail$ à C.
3. Nous ajoutons la clause $cinema \vee travail$, qui est le résultat de la résolution de $beau \vee cinéma \vee travail$ et $\neg beau$, à l'ensemble C.
4. Nous appliquons la règle de résolution aux clauses $\neg paresseux \vee \neg travail$ et $paresseux$ et nous ajoutons la clause résultante $\neg travail$ à C.
5. Nous ajoutons $cinéma$ à C après avoir fait la résolution de $cinema \vee travail$ et $\neg travail$.
6. Nous avons maintenant $C =$

   $\{\neg beau \vee plage, beau \vee cinéma \vee travail, \neg paresseux \vee \neg travail, \neg beau, paresseux, plage \vee cinéma \vee travail, cinema \vee travail, \neg travail, cinéma\}$

   Comme il ne reste plus de paires de clauses auxquelles nous pouvons appliquer la règle de résolution, nous passons à l'étape suivante.

7. Nous supprimons $\neg beau \vee plage$ de C car la clause $\neg beau$ est plus forte. Nous supprimons $\neg paresseux \vee \neg travail$ parce que C contient la clause $\neg travail$. Enfin, nous supprimons les clauses $beau \vee cinéma \vee travail$, $plage \vee cinéma \vee travail$, et $cinéma \vee travail$ à cause de la clause $cinéma$ qui est plus spécifique.
8. Nous renvoyons l'ensemble $C =$

   $\{\neg beau, paresseux, \neg travail, cinéma\}$

L'ensemble des impliqués premiers de ϕ contiennent exactement les mêmes informations que ϕ mais elle à l'avantage d'être beaucoup plus lisible.

Chapitre 6: Apprentissage Automatique

Lors des derniers chapitres nous avons évoqué l'importance pour un agent intelligent d'avoir des connaissances relatives à son environnement. L'objectif du ce chapitre est d'introduire le domaine de l'apprentissage automatique qui s'intéresse à la façon dont l'agent peut arriver à acquérir ces connaissances.

Introduction à l'apprentissage

L'apprentissage automatique étudie les méthodes permettant à un agent de construire de nouvelles connaissances à partir de son expérience. Nous pouvons distinguer trois classes générales de problèmes d'apprentissage selon les types d'informations dont dispose l'agent :

Apprentissage supervisé L'objectif de l'apprentissage supervisé est d'apprendre à classer des instances à partir d'un ensemble d'exemples qui sont étiquetés par leurs classes (par un expert humain) [1]. Un exemple typique de l'apprentissage supervisé serait donc la reconnaissance visuelle des chiffres : on présente à l'agent un ensemble de chiffres manuscrits (étiquetté par les nombres qu'ils représentent) et il doit apprendre à identifier les différents chiffres. Les algorithmes principaux pour l'apprentissage supervisé sont les *réseaux de neurones*, les *machines à vecteur de support*, et les *arbres de décision*. Cette dernière méthode sera examinée plus en détail dans la deuxième partie du chapitre.

Apprentissage non-supervisé L'inconvénient majeur de l'apprentissage supervisé est que les données doivent être étiquetées. Or, dans le monde réel, les données ne le sont généralement pas, donc l'étiquetage doit se faire à la main, ce qui requiert un temps considérable de la part d'un expert humain. En apprentissage non-supervisé, nous ne reçevons que les données brutes, et l'objectif est de trouver des points communs de ces données. Les applications possibles de l'apprentissage non-supervisé

1. Ou, plus généralement, d'apprendre une fonction f à partir un ensemble d'exemples de paires $(x, f(x))$.

sont très nombreuses car aujourd'hui nous disposons de plus en plus de données et la difficulté est d'analyser ces données et d'en extraire les informations intéressantes. Nous pourrions par exemple utiliser l'apprentissage non-supervisé dans la séismologie pour trouver les zones à risque pour les tremblements de terre ou bien dans le marketing pour analyser les comportements des consommateurs. Deux algorithmes parmi les plus connus pour faire de la classification sont *la méthode des k-moyennes* et *la méthode des sommes pondérées de gaussiennes*.

Apprentissage par renforcement En apprentissage par renforcement, l'agent doit apprendre à bien agir dans son environnement afin de maximiser sa récompense. A chaque instant, l'agent peut percevoir (complètement ou partiellement) son environnement et choisit ensuite une action à effectuer. Après chaque action, il reçoit une récompense (qui peut être positif, négatif, ou zéro). A partir de ce feedback et de ses perceptions, l'agent doit apprendre un comportement qui l'amène à obtenir le plus de récompense possible. Un exemple d'un problème d'apprentissage par renforcement pourrait être d'apprendre à jouer aux échecs (la récompense serait de 0 si le jeu n'est pas fini ou si c'est un match null, $+1$ si l'agent gagne, et -1 pour une perte). Un autre exemple pourrait être d'apprendre à un robot à se deplacer plus vite (ici nous pourrions donner une récompense positive chaque fois le robot couvre une certaine distance). L'apprentissage par renforcement a été utilisé avec succès sur de nombreux problèmes et en particulier pour apprendre aux robots des mouvements physiques complexes. Les algorithmes les plus utilisés pour l'apprentissage par renforcement sont les *algorithmes de différence temporelle*[2].

Ces trois types d'apprentissage sont complémentaires, et le choix de la bonne méthode d'apprentissage dépend du problème en question. Si nous voulons faire apprendre à un agent une tâche spécifique dont nous pouvons fournir des exemples, l'apprentissage supervisé serait un bon choix. S'il s'agit d'apprendre un comportement compliqué, l'apprentissage par renforcement serait probablement plus approprié. Enfin, si nous n'avons pas d'objectif clair, nous choisirons l'apprentissage non-supervisé pour découvrir des régularités dans un ensemble de données.

2. C'était un algorithme de la différence temporelle qui a été utilisé pour developper le réseau de neurones au coeur du programme TD-GAMMON (qui joue au backgammon) que nous avons mentionné au chapitre 4.

Méthode des arbres de décision

Dans cette section, nous allons introduire la méthode des arbres de décision qui est l'un des algorithmes les plus populaires pour l'apprentissage supervisé. Cette méthode consiste en la construction d'un *arbre de décision* à partir d'un ensemble d'exemples. Cet arbre est ensuite utilisé pour classer de nouvelles instances. L'avantage principal de cette approche est sa simplicité : les arbres de décision sont facilement interpétables par des humains, ce qui n'est pas le cas pour les réseaux de neurones ou les machines à vecteurs de supports.

Commençons d'abord par une définition plus formelle de l'apprentissage supervisé. En entrée, nous reçevons un multi-ensemble [3] d'exemples $E \subset A_1 \times ... \times A_n \times C$ qui sont décrits par un certain nombre d'attributs $A_1, ..., A_n, C$, dont le dernier est l'attribut cible. Chaque attribut A a un domaine $dom(A)$ de valeurs possibles. En général, les domaines pourraient avoir infiniment de valeurs, mais dans cette section, nous ne considérons que les attributs avec les domaines finis. En sortie, nous devrons fournir une fonction $f : A_1 \times ... \times A_n \rightarrow C$ qui dit comment classer tous les éléments de $A_1 \times ... \times A_n$. Bien entendu, certaines fonctions sont plus satisfaisantes que d'autres. Idéalement, nous aimerions trouver la fonction f qui donne la bonne réponse pour toutes les éléments de $A_1 \times ... \times A_n$.

Considérons maintenant un exemple concret. Dans cet exemple, nous cherchons à détecter si un mail est un spam ou non. Nous disposons des données suivantes :

	Auteur	**MotClés**	**HTML**	**Majuscule**	**Spam**
1	*connu*	*oui*	*oui*	*non*	*non*
2	*inconnu*	*non*	*non*	*oui*	*oui*
3	*inconnu*	*oui*	*non*	*non*	*non*
4	*inconnu*	*oui*	*oui*	*oui*	*oui*
5	*connu*	*non*	*non*	*non*	*non*
6	*inconnu*	*non*	*oui*	*oui*	*oui*
7	*connu*	*non*	*non*	*oui*	*non*
8	*inconnu*	*oui*	*oui*	*non*	*oui*
9	*connu*	*oui*	*non*	*non*	*non*
10	*inconnu*	*oui*	*non*	*oui*	*oui*
11	*connu*	*oui*	*non*	*non*	*non*
12	*inconnu*	*non*	*non*	*non*	*non*

3. Un *multi-ensemble* est semblable à un ensemble, sauf que des éléments peuvent appaitre plusieurs fois. $\{a, b, a\}$ serait donc un exemple d'un multi-ensemble qui n'est pas un ensemble. Pour simplicité, pour le reste du chapitre, nous dirons "ensemble" au lieu de "multi-ensemble".

Ici nous avons les quatre attributs **Auteur** (si l'auteur du mail est connu ou non [4]), **MotClés** (si le message contient l'un d'un ensemble de mots associé aux spams), **HTML** (si le mail contient un lien html), et **Majuscule** (si le message contient des mots en majuscules ou non) plus l'attribut cible **Spam**. Les domaines des attributs sont : $dom(\textbf{Auteur}) = \{connu, inconnu\}$, $dom(\textbf{MotClés}) = \{oui, non\}$, $dom(\textbf{HTML}) = \{oui, non\}$, $dom(\textbf{Majuscule}) = \{oui, non\}$, et $dom(\textbf{Spam}) = \{oui, non\}$.

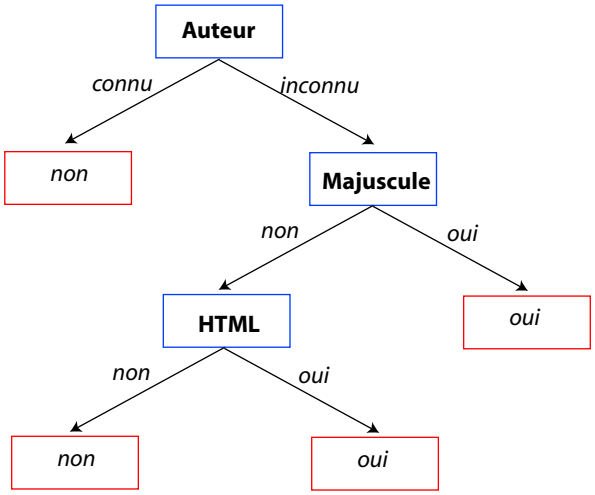

FIGURE 1 – Un arbre de décision qui classe correctement tous les exemples.

La Figure 1 montre un arbre de décision qui permet de bien classer les données dans notre exemple. Comme dans tous les arbres de décision, les noeuds non-terminaux sont étiquetés par des attributs (non-cibles), les arcs sont étiquetés par une valeur dans le domaine de l'attribut associé au noeud parent, et chaque feuille de l'arbre prend une valeur dans le domaine de l'attribut cible.

Un arbre de décision définit la valeur de l'attribut cible pour chaque instance possible. Pour trouver la valeur d'une instance spécifique, il suffit de parcourir l'arbre en choisissant l'arc qui correspond à l'instance en question jusqu'à arriver à une feuille dont l'étiquette donne la classe de l'instance.

4. Nous n'avons pas précisé comment définir un auteur connu. Une possibilité serait de marquer un auteur comme connu s'il a été le destinataire d'un mail de cette messagerie ou si l'adresse de l'auteur se trouve dans le carnet d'adresses.

Prenons par exemple l'instance (*inconnu*, *non*, *oui*, *non*). Nous commençons à la racine qui correspond à l'attribut **Auteur**. Comme notre instance prend la valeur *inconnu* pour l'attribut **Auteur**, nous suivons l'arc droit. Puis comme le prochain attribut est **Majuscule** et notre instance a la valeur *non* pour **Majuscule**, nous prenons l'arc gauche. Nous sommes maintenant à un noeud qui correspond à l'attribut **HTML**, donc nous choisissons l'arc droit (dont la valeur est *oui*) et nous trouvons une feuille marquée *oui*. L'instance (*inconnu*, *non*, *oui*, *non*) est donc classée comme un mail spam par notre arbre de décision.

A chaque ensemble d'exemples correspond plusieurs arbres de décisions. Comment donc savoir si un certain arbre de décision est un bon choix pour un ensemble de données ? Un premier critère est le *taux de couverture* de l'arbre, i.e. le pourcentage des exemples qui sont bien classés par l'arbre. L'importance de cette critère devrait être évidente car les exemples dont nous disposons sont nos seuls informations sur la fonction à apprendre. Nous voulons donc que notre arbre capte les informations dans ces exemples.

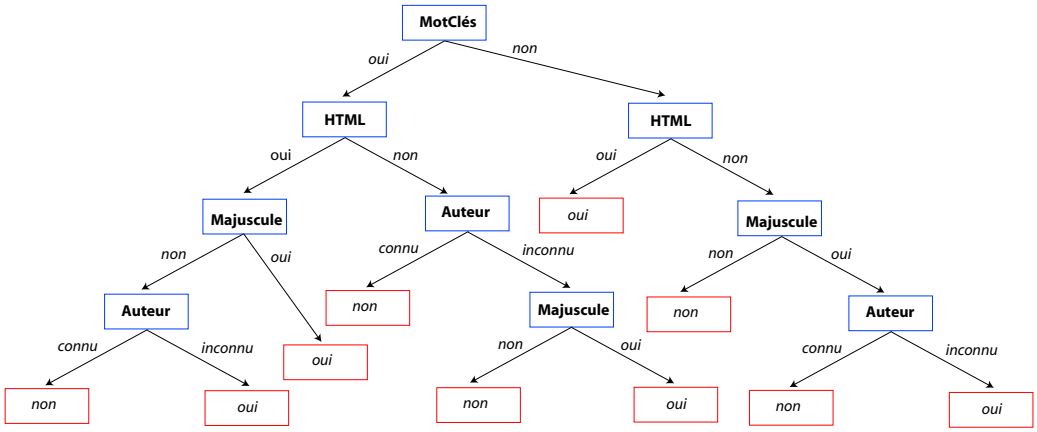

FIGURE 2 – Un arbre de décision plus compliqué.

Considérons maintenant l'arbre de décision de la Figure 2. Cet arbre a exactement le même taux de couverture (100%) que celui de la Figure 1. Or, nous avons tendance à dire que ce deuxième arbre est moins satisfaisant que le premier. Ceci laisse à croire que ce n'est pas que le taux de couverture qui compte mais aussi la *simplicité* de l'arbre (qui peut être mesurée par le nombre de noeuds non-terminaux). Il y a (au moins) deux bonnes raisons de

préférer les arbres simples. D'un côté, les arbres simples sont beaucoup plus faciles à comprendre, ce qui n'est pas négligeable vu que l'interprétabilité des arbres de décision est l'un des avantages principaux de cette approche. D'un autre côté, les arbres simples avec des taux de couverture raisonnables donnent souvent de meilleurs résultats sur des instances nouvelles que les arbres complexes qui couvrent plus d'exemples. Cela s'explique par le fait que les arbres simples ne nous laissent pas construire des fonctions trop compliquées. Nous ne pouvons donc que tenir compte des règles importantes dans les données et non pas des petits détails. Or, avec les arbres complexes nous pouvons construire des fonctions beaucoup plus compliquées, avec une plus forte tendance à effectuer une simple mémorisation d'exemples. Nous obtenons ainsi des arbres qui donnent une performance optimale sur les exemples déjà vus mais une performance beaucoup moins satisfaisante sur de nouvelles instances. Ce phénomène, qui s'appelle *surapprentissage*, est un problème très répandu en apprentissage supervisé, et arrive aussi pour les réseaux de neurones .

Comment construire des arbres de décision qui ont à la fois un bon taux de couverture et un petit nombre de noeuds ? Il n'y a pas de solution miracle [5], mais plusieurs heuristiques ont été dévéloppées qui permettent de choisir parmi les différents attributs. Ici, nous introduisons le *gain d'information*, qui est probablement l'heuristique la plus connue pour les arbres de décision.

Nous commençons par introduire la notion d'*entropie* qui est une quantité qui mesure l'information (et donc nous sera utile pour évaluer les *gains* d'information). En gros, l'entropie d'un ensemble d'éléments étiquetés mesure combien les différentes classes sont mélangées. L'entropie d'un ensemble D est minimale quand tous les exemples contenus dans D appartiennent à la même classe. Elle est maximale quand les exemples de D sont uniformément répartis parmi les différentes classes. Si nous avons n classes différentes, l'entropie d'un ensemble D est définie par :

$$H(D) = -(p_1 \log_2 p_1 + p_2 \log_2 p_2 + ... + p_n \log_2 p_n)$$

où p_i donne la proportion d'éléments dans D qui sont étiquetés par la i-ème classe.

Le *gain d'information* d'un attribut par rapport à A représente la différence entre l'entropie avant et après le branchement sur A. Plus le gain est important, plus l'attribut est utile pour séparer les exemples en classes. Nous

5. Il a été prouvé que trouver un arbre de décision minimal qui est consistant avec un ensemble d'exemples est un problème NP-complet.

pouvons calculer le gain d'information avec la formule suivante :

$$G(A, D) = H(D) - \left(\Sigma_{x \in dom(A)} \frac{|D[A = x]|}{|D|} H(D[A = x])\right)$$

où $D[A = x]$ signifie : les éléments de D ayant la valeur x pour l'attribut A.

Nous présentons maintenant un algorithme ARBRE qui utilise le gain d'information pour construire les arbres de décision. Le principe de l'algorithme est très simple : nous choisissons toujours l'attribut qui a le meilleur gain d'information, nous faisons le branchement sur les différentes valeurs dans le domaine de l'attribut, puis nous appelons l'algorithme à nouveau pour choisir les prochains attributs. Nous nous arrêtons quand tous les exemples d'une branche ont la même étiquette ou bien quand il ne reste pas d'attributs à tester (i.e. la branche contient déjà tous les attributs). L'algorithme ne garantit pas que l'arbre produit soit le plus petit pour son taux de couverture, mais en pratique il produit des arbres de décisions plutôt satisfaisants.

fonction ARBRE($\mathcal{A}$, C, D)
  créer un noeud racine R
  si tous les éléments de D ont l'étiquette c **alors**
    étiqueter R par c
  sinon si $\mathcal{A}$ est vide
    étiqueter R avec la valeur de C la plus fréquente dans D
  sinon
    étiqueter R par l'attribut $A \in \mathcal{A}$ avec le plus grand gain d'information
    pour chaque $a \in dom(A)$ **faire**
      ajouter un arc avec étiquette a
      si $D[A = a] = \emptyset$
        ajouter à la fin de l'arc un noued étiqueté par la valeur de C
                la plus fréquente dans D
      sinon
        mettre l'arbre ARBRE($\mathcal{A} \setminus A$, C, $D[A = a]$) à la fin de la branche
  retourner l'arbre R

FIGURE 3 – Un algorithme pour générer les arbres de décision. Les arguments de l'algorithme sont un ensemble de attributs $\mathcal{A}$ qui ne sont pas encore utilisés, l'attribut cible C, et un (multi)-ensemble non-vide D d'exemples étiquetés par des valeurs de $dom(C)$.

Nous illustrons le fonctionnement de cet algorithme en l'appliquant à notre exemple. Nous appelons ARBRE avec $\mathcal{A} = \{$**Auteur**, **MotClés**, **HTML**,

Majuscule}, $C = \textbf{Spam}$, et D les douzes exemples ci-dessus. Comme les exemples ne sont pas tous dans la même classe, et $\mathcal{A}$ n'est pas vide, nous devrons calculer l'attribut avec le plus grand gain d'information. Commençons d'abord par l'attribut **Auteur**. Tout d'abord il nous faut trouver l'entropie de D :

$$H(D) = -(\frac{5}{12}\log_2\frac{5}{12} + \frac{7}{12}\log_2\frac{7}{12}) = 0,98$$

Ensuite, il faut calculer les entropies pour les différentes valeurs de **Auteur**. Pour la valeur *connu*, tous les exemples ont l'étiquette *non*, donc nous obtiendrons une entropie de 0 :

$$H(D[\textbf{Auteur} = connu]) = -(\frac{0}{5}\log_2\frac{0}{5} + \frac{5}{5}\log_2\frac{5}{5}) = 0$$

Pour *inconnu*, il y a 5 exemples *oui* et 2 exemples *non*, ce qui nous donne une valeur de $0,86$:

$$H(D[\textbf{Auteur} = inconnu]) = -(\frac{5}{7}\log_2\frac{5}{7} + \frac{2}{7}\log_2\frac{2}{7}) = 0,86$$

Nous trouvons un gain d'information de $0,48$ pour l'attribut **Auteur** :

$$G(\textbf{Auteur}, D) = 0,98 - (\frac{5}{12} * 0 + \frac{7}{12} * 0,86) = 0,48$$

Cherchons maintenant les gains d'informations des autres attributs. Prenons d'abord **MotClés** :

$$H(D[\textbf{MotClés} = oui]) = -(\frac{3}{7}\log_2\frac{3}{7} + \frac{4}{7}\log_2\frac{4}{7}) = 0,99$$

$$H(D[\textbf{MotClés} = non]) = -(\frac{2}{5}\log_2\frac{2}{5} + \frac{3}{5}\log_2\frac{3}{5}) = 0,97$$

$$G(\textbf{MotClés}, D) = 0,98 - (\frac{7}{12} * 0,99 + \frac{5}{12} * 0,97) = 0$$

Considérons ensuite l'attribut **HTML** :

$$H(D[\textbf{HTML} = oui]) = -(\frac{3}{4}\log_2\frac{3}{4} + \frac{1}{4}\log_2\frac{1}{4}) = 0,81$$

$$H(D[\textbf{HTML} = non]) = -(\frac{2}{8}\log_2\frac{2}{8} + \frac{6}{8}\log_2\frac{6}{8}) = 0,81$$

$$G(\textbf{HTML}, D) = 0,98 - (\frac{4}{12} * 0,81 + \frac{8}{12} * 0,81) = 0,17$$

Terminons avec l'attribut **Majuscule** :

$$H(D[\textbf{Majuscule} = oui]) = -(\frac{4}{5}\log_2\frac{4}{5} + \frac{1}{5}\log_2\frac{1}{5}) = 0,72$$

$$H(D[\textbf{Majuscule} = non]) = -(\frac{1}{7}\log_2\frac{1}{7} + \frac{6}{7}\log_2\frac{6}{7}) = 0,59$$

$$G(\textbf{Majuscule}, D) = 0,98 - (\frac{5}{12} * 0,72 + \frac{7}{12} * 0,59) = 0,33$$

C'est donc **Auteur** qui a le plus grand gain d'information, donc nous utilisons cet attribut pour commencer notre arbre. Comme tous les exemples ayant **Auteur**=*connu* ont tous la même valeur (=*non*), nous ajoutons un arc marqué *connu* et un noeud étiquetté par *non*. Pour **Auteur**=*inconnu*, nous devrons appliquer l'algorithme à nouveau pour choisir un attribut pour cette branche de l'arbre.

Nous travaillons maintenant sur les exemples dont l'auteur est inconnu, i.e. nous mettons $D = D[\textbf{Auteur} = inconnu]$. Nous avons déjà calculé l'entropie de cet ensemble : $H(D[\textbf{Auteur} = inconnu]) = 0,86$. Maintenant il faut déterminer les gains d'information respectifs pour les trois attributs restants. Commençons par **MotClés** :

$$H(D[\textbf{Auteur} = inconnu][\textbf{MotClés} = oui]) = -(\frac{3}{4}\log_2\frac{3}{4} + \frac{1}{4}\log_2\frac{1}{4}) = 0,81$$

$$H(D[\textbf{Auteur} = inconnu][\textbf{MotClés} = non]) = -(\frac{2}{3}\log_2\frac{2}{3} + \frac{1}{3}\log_2\frac{1}{3}) = 0,92$$

$$G(\textbf{MotClés}, D[\textbf{Auteur} = inconnu]) = 0,86 - (\frac{4}{7} * 0,81 + \frac{3}{7} * 0,92) = 0,01$$

Passons ensuite à l'attribut **HTML** :

$$H(D[\textbf{Auteur} = inconnu][\textbf{HTML} = oui]) = -(\frac{3}{3}\log_2\frac{3}{3} + \frac{0}{3}\log_2\frac{0}{3}) = 0$$

$$H(D[\textbf{Auteur} = inconnu][\textbf{HTML} = non]) = -(\frac{2}{4}\log_2\frac{2}{4} + \frac{2}{4}\log_2\frac{2}{4}) = 1$$

$$G(\textbf{HTML}, D[\textbf{Auteur} = inconnu]) = 0,86 - (\frac{3}{7} * 0 + \frac{4}{7} * 1) = 0,29$$

Finissons avec **Majuscule** :

$$H(D[\textbf{Auteur} = inconnu][\textbf{Majuscule} = oui]) = -(\frac{4}{4}\log_2\frac{4}{4} + \frac{0}{4}\log_2\frac{0}{4}) = 0$$

$$H(D[\textbf{Auteur} = inconnu][\textbf{Majuscule} = non]) = -(\frac{1}{3}\log_2\frac{1}{3} + \frac{2}{3}\log_2\frac{2}{3}) = 0,92$$

$$G(\textbf{Majuscule}, D[\textbf{Auteur} = inconnu]) = 0,86 - (\frac{4}{7} * 0 + \frac{3}{7} * 0,92) = 0,47$$

C'est donc l'attribut **Majuscule** qui a le meilleur gain d'information. Nous étiquetons donc notre noeud par **Majuscule**, et nous ajoutons deux arcs, l'un pour **Majuscule**=*oui* et l'autre pour **Majuscule**=*non* Les exemples ayant **Majuscule**=*oui* ont tous la valeur *oui*, donc nous ajoutons un noeud avec l'étiquette *oui*. Pour **Majuscule**=*non*, nous avons toujours des exemples avec des étiquettes différentes, donc nous continuons.

Maintenant nous ne considérons que les exemples de D ayant **Auteur** = *inconnu* et **Majuscule**=*non*. L'entropie de cet ensemble a été déterminée auparavant : $H(D[\textbf{Auteur} = inconnu][\textbf{Majuscule} = non]) = 0,92$. Il nous reste que deux attributs possibles : **HTML** et **MotClés**. Les gains d'information pour ces deux attributs sont (je vous laisse le calcul en exercice) :

$$G(\textbf{HTML}, D[\textbf{Auteur} = inconnu][\textbf{Majuscule} = non]) = 0,92$$

$$G(\textbf{MotClés}, D[\textbf{Auteur} = inconnu][\textbf{Majuscule} = non]) = 0,25$$

Nous donnons donc l'étiquette **HTML** à notre noeud, et nous créons deux noeuds fils, l'un pour le cas **HTML**=*oui* (qui aura l'étiquette *oui*) et **HTML** = *non* (avec l'étiquette *non*). L'arbre construit par l'algorithme est donc celui de Figure 1.

L'algorithme que nous avons présenté ici est l'algorithme classique pour construire les arbres de décision, et de nombreuses extensions ou modifications existent. D'un côté, il y a eu des extensions de cette approche aux problèmes ayant des attributs dont les domaines sont numériques (e.g. **Salaire**, **Poids**, **Température**). Il y a eu également des extensions pour prendre en compte des exemples dont les valeurs de certains des attributs ne sont pas connus (e.g. (*inconnu*, ?, *oui*, *non*), où la valeur de **MotClés** n'est pas spécifiée). D'un autre côté, il y a eu des travaux visant à améliorer la qualité des arbres produits. Une possibilité simple est de ne pas continuer à construire une branche si aucun attribut ne donne un gain d'information significatif. Une autre méthode consiste à ajouter une dernière étape à l'algorithme où certaines parties de l'arbre sont supprimées, afin de trouver un arbre plus simple. Une troisième possibilité serait de produire plusieurs arbres de décision différents et de les utiliser tous ensemble pour classer les instances. Cette dernière méthode peut s'avérer efficace, mais les familles d'arbres produites sont plus difficiles à comprendre que les arbres uniques produits par l'algorithme classique.