Introduction à l'intelligence artificielle à destination d'étudiants de licence informatique en troisième année

—

Première partie

Jean Lieber, Université de Lorraine, Jean.Lieber@loria.fr

Dernière version : 10 janvier 2019
Page du cours : http://homepages.loria.fr/JLieber/cours/iia/

Remarques sur ce document

— Ce document rassemble des notes de cours utiles pour les intervenants et pour les étudiants mais ne constitue pas un polycopié complet. Il ne se substitue donc pas à des prises régulières de notes durant les cours.

— Ce document contient probablement des erreurs (d'orthographe, de grammaire ou d'autres natures). Merci de le signaler à son auteur (par exemple, à la fin d'un cours).

Remerciements

L'auteur de ces notes de cours tient à remercier Laurent Bougrain pour ses conseils et les notes de cours qu'il lui a donné.

Sommaire

*****

1 Introduction

1.1 Brefs aperçus sur l'intelligence artificielle

Le terme d'***intelligence artificielle*** (abrégé ***IA*** dans la suite du document) recouvre des principes, méthodes et techniques d'informatique pour donner à une machine (un ordinateur, un robot, etc.) des comportements considérés à un instant donné comme intelligents. La question de savoir ce qui relève de l'IA et ce qui n'en relève pas n'est pas simple et fait l'objet de débats. Calculer une valeur approchée de $\sqrt{10}$ est un problème qui requiert de l'intelligence, pour un être humain, par conséquent une simple calculette ne fait-elle pas de l'IA ? La réponse actuelle est plutôt négative, mais cela peut se discuter.

L'aperçu de l'IA présenté ci-dessous est très approximatif, pour sa chronologie, et très incomplète, pour son contenu. Ce dernier sera également déséquilibré du fait de la spécialité de l'auteur de ces notes (qui assume cette subjectivité).

L'IA fait partie des ***sciences cognitives*** qui regroupe les disciplines scientifiques intéressées par les phénomènes de la cognition (psychologie cognitive, philosophie de l'esprit, neurosciences, logique, etc.). Une question qui se pose depuis l'origine de l'IA (vers les années 1950) et même avant est la suivante : est-il possible de doter une machine de capacités qui la rendent aussi intelligente qu'un être humain ? Les partisans de l'« IA forte » considère que la réponse à cette question est positive, mais la preuve n'a pas encore été faite et certains (les détracteurs de l'IA forte) pensent que c'est impossible.

Alan Turing, un des « pères » de l'informatique, a abordé cette question dans son article *Computing Machinery and Intelligence* (1950, Mind). Dans cet article, un test — désormais appelé *test de Turing* — consiste à confronter un humain avec une machine se faisant passer pour un humain via un système de dialogue écrit. Une machine qui passerait le test de Turing arriverait à tromper l'être humain. Même si l'IA a beaucoup évolué, on ne trouve pas de machine qui réussisse pleinement le test de Turing.

Une limite de ce test serait qu'une machine intelligente ne serait pas nécessairement capable de réussir ce test, de la même façon qu'on arrive à faire des machines volantes mais pas de machines parfaitement à même d'imiter le vol d'un oiseau.

Les partisans de l'« IA faible » ont un objectif plus pragmatique : simuler/modéliser des comportements intelligents dans une machine à fin d'applications utiles.

L'IA s'est développée parallèlement à l'informatique et son essor s'est vraiment fait dans les années 1980. À cette époque, on parlait beaucoup de systèmes experts et de jeux. C'est ce qu'on appelle parfois la ***GOFAI*** (*good old-fashioned artificial intelligence*, « bonne vieille IA » en français). Les systèmes experts peuvent être rapprochés de représentations logiques, telles que celles permises par le langage Prolog. Ce cours concerne essentiellement l'autre partie de la GOFAI (qui concerne notamment certains types de jeux).

À la fin des années 1980, l'IA a connu une crise : les systèmes experts (ou systèmes à base de règles) n'ont pas tenu toutes leurs promesses. Il semble en effet qu'acquérir des connaissances sous forme de règles (*si . . . alors . . .*) ne soit pas aisé pour tous les problèmes : les règles deviennent de plus en plus difficiles à acquérir, au fur et à mesure qu'on entre dans des cas plus particuliers, et de plus en plus difficiles à appréhender. Le *problème de la qualification*, un des problèmes envisagé par John McCarthy (l'inventeur du terme *artificial intelligence*), illustre cela. Il dit qu'il est en pratique très difficile d'énumérer toutes les conditions rendant une action possible. Par exemple, si on doit donner les conditions pour qu'une voiture à essence démarre, si on en trouve facilement quelques unes, il est peu probable qu'on pense à la condition « Il ne faut pas qu'il y ait de pomme de terre dans le carburateur. » !

Dans les années 1990, l'IA a connu moins de succès visibles mais a continué à être étudié : en fait, beaucoup d'applications avait une composante IA, mais ce n'était plus vraiment un argument de vente. La distinction entre ***IA numérique*** et ***IA symbolique*** s'est beaucoup affirmée dans ces années (et encore à présent).

L'***IA numérique*** utilise des représentations souvent simples (p. ex., des n-uplets de réels). Elle comprend notamment des approches issues des statistiques (p. ex., les modèles stochastiques) et des approches bio-inspirées, parmi lesquelles les réseaux de neurones artificiels (notamment les réseaux de neurones « profonds » qui ont beaucoup de succès ces dernières années). La ***reconnaissance des formes*** (p. ex., la reconnaissance de la parole ou la vision artificielle) utilise souvent des techniques d'IA numérique. L'IA numérique s'appuie beaucoup sur le concept d'***émergence*** : à partir d'éléments simples mais nombreux (p. ex., des neurones artificiels), on fait émerger des comportements complexes. Certains systèmes multi-agents (SMA) relèvent également de l'émergence : dans ce cadre un agent est un objet ayant un comportement, des contraintes et des objectifs autonomes et une colonie d'agents atteint des objectifs globaux et complexes. On parle d'agents réactifs.

Les agents cognitifs sont plus complexes et relèvent davantage de l'***IA symbolique***. À partir des années 1990, le développement de celle-ci peut être largement considéré comme une réponse aux limitations des systèmes experts. À la différence des systèmes d'IA numérique, ces systèmes manipulent des *connaissances explicites*, on parle de ***systèmes à base de connaissances (explicites)***. Un des problèmes des systèmes experts est celui du « goulot d'étranglement de l'acquisition des connaissances ». Pour y répondre, plusieurs pistes complémentaires ont été envisagées.

Parmi ces pistes, plusieurs proposent des approches pour mieux obtenir ces connaissances. L'***acquisition de connaissances*** est un ensemble de méthodologie pour l'acquisition et la modélisation des connaissances par dialogues avec des experts. Ce domaine a évolué vers ce qu'on appelle aujourd'hui l'***ingénierie des connaissances*** qui, au-delà de la seule acquisition, s'intéresse aux problèmes de la maintenance et de l'évolution des connaissances.

L'***apprentissage artificiel symbolique*** réunit des techniques d'apprentissage artificiel qui débouchent sur des bases de connaissances (bases de règles ou autres types de bases). Souvent, cet apprentissage se fait par généralisation inductive à partir d'exemples.

L'***extraction de connaissances dans des bases de données*** est proche de l'apprentissage artificiel. Une différence est que l'accent est mis davantage sur des algorithmes qui peuvent travailler sur des grandes masses de données et sur l'intervention d'un analyste (expert du domaine des données) et d'un ingénieur des connaissances pour piloter le processus.

Une autre piste consiste à utiliser des connaissances plus factuelles que des règles : le ***raisonnement à partir de cas*** (RÀPC) consiste à résoudre des problèmes à partir d'épisodes de résolution de problèmes particuliers, appelés cas. Si on le compare avec un raisonnement à partir de règles (RÀPR), la différence se situe à deux niveaux. D'abord le RÀPC *adapte* les cas aux nouvelles situations alors que le RÀPR *applique* les règles dans ces situations (de façon déductive). Ensuite, l'acquisition des cas est réputée plus facile que l'acquisition des règles (il « suffit » de représenter des expériences particulières) et les cas sont plus faciles à appréhender que les règles.

La ***représentation des connaissances*** étudie des logiques destinées à être manipulées dans des systèmes à base de connaissances symboliques et des inférences non nécessairement déductives dans ces formalismes. C'est une problématique qui a été abordée lors du cours de logique, au premier semestre de cette troisième année de licence.

De façon un peu rapide, les différences entre IA numérique et IA symbolique sont les suivantes. L'IA numérique utilise plus les mathématiques continues (analyse, étude de fonctions sur $\mathbb{R}^n$, dérivées, algèbre linéaire, etc.) alors que l'IA symbolique utilise plus les mathématiques discrètes (graphes, algèbres finies, logique). Les sciences cognitives plus proches de l'IA numérique sont la biologie et les neurosciences alors que celles qui sont plus proches de l'IA symbolique sont la logique, la philosophie de l'esprit et la psychologie cognitive. Cependant, cette distinction entre ces deux types de l'IA ne se veut pas une barrière et certains travaux et domaines de l'IA peuvent relever des deux types.

1.2 Plan du cours

Ce cours relève donc essentiellement de la GOFAI, même si les notions relatives aux systèmes experts (qui ont plus leur place dans un cours de logique et ... de représentation des connaissances) ne seront pas abordées.

On y verra des parcours d'arbres (finis et infinis) ce qui justifie un rappel sur les parcours classiques d'arbres représentés explicitement (section 2).

La section 3 décrit le problème de parcours d'espaces d'états et peut être vu comme le parcours (généralement non exhaustif) d'arbres potentiellement infinis. Il permet de résoudre des problèmes discrets sans hasard, tels que certains jeux (le taquin, le cube de Rubik, etc.) et a d'autres applications moins ludiques (mais non moins passionnantes).

La section 4 décrit certains types de jeux à deux joueurs. Son objectif est de décrire des algorithmes pour qu'une machine puisse jouer contre un humain à ces jeux.

1.3 Informations pratiques

Le cours est constitué de :
— 10 heures de cours, dont 5 heures sur cette première partie ;
— 10 heures de travaux dirigés, dont 6 heures sur cette première partie ;
— 10 heures de travaux pratiques ;
— Le nombre d'heures suffisant en travail personnel pour assimiler avec précision les notions du cours et pour préparer les séances de TD et les séances de TP.

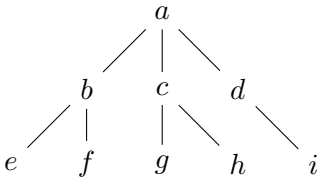
Les modalités de contrôle de connaissances de cette UE sont comme suit :

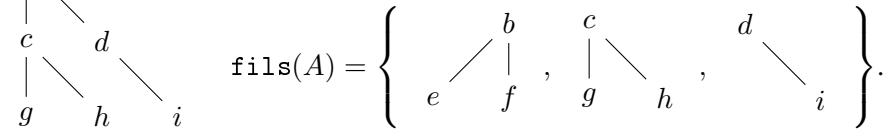
$$\begin{aligned} n_{\text{TP}} &= \text{note de TP} \\ n_1 &= \text{note de l'examen (sur copies) de la première session} \\ n_2 &= \text{note de l'examen (sur copies) de la deuxième session} \\ \text{note de la première session} &= 0{,}4\, n_{\text{TP}} + 0{,}6\, n_1 \\ \text{note de la deuxième session} &= 0{,}4\, n_{\text{TP}} + 0{,}6\, n_2 \end{aligned}$$

*****

2 Rappels : parcours d'arbres

Dans cette section, les opérations d'accès à un arbre sont :
— `racine`, qui à un arbre A associe la racine de A ($\texttt{racine}(A)$) ;
— `fils`, qui à un arbre A associe le *multi-ensemble* des arbres fils de A ($\texttt{fils}(A)$).

Par exemple, si $A =$ 

alors $\texttt{racine}(A) = a$ et
$\texttt{fils}(A) =$ 

.

Un ***nœud*** (ou sous-arbre) d'un arbre est soit l'arbre lui-même, soit ses fils, soit ses petits-fils, etc. Le ***rang*** r d'un nœud N dans un arbre A est le nombre d'arcs qui le sépare de A : le rang de A dans A est 0, le rang des fils de A dans A est 1, le rang des petits-fils de A dans A est 2, etc. La ***profondeur*** d'un arbre est le maximum des rangs de cet arbre.

On notera qu'on considère un multi-ensemble de fils, et pas une liste : dans ce cours, l'ordre sur les fils n'est pas pris en compte (mais quand on programmera, on pourra utiliser une liste pour représenter un multi-ensemble).

Faire le parcours d'un arbre c'est considérer ses nœuds dans un certain ordre et, pour chaque nœud, effectuer un traitement (par exemple, l'affichage de la racine du nœud).

Un ***parcours en profondeur*** d'un arbre consiste à parcourir les branches de cet arbre. Si ce parcours est préfixé — ce qu'on considérera dans la suite — alors on fera le parcours d'une branche de la racine vers la feuille. Techniquement, il suffit alors de traiter l'arbre principal (p. ex., en affichant sa racine) puis faire un appel récursif

```
Procédure parcours_profondeur_infixé (A : arbre)
Variable
  | N : arbre
Début
  | Traiter A (p. ex., afficher(racine(A)))
  | Pour N ∈ fils(A) Faire
  |   | parcours_profondeur_infixé(N)
  | Finpour
Fin
```

FIGURE 1 – Parcours en profondeur d'un arbre.

```
Procédure parcours_largeur (A : arbre)
Variables
  | OUVERTS : file (d'arbres)
  | courant : arbre
Début
  | OUVERTS ← enfiler(A, file_vide)
  | Tant que non est_vide(OUVERTS) Faire
  |   | courant ← premier(OUVERTS)
  |   | Traiter courant (p. ex., afficher(racine(courant)))
  |   | OUVERTS ← défiler(OUVERTS)      /* enlever courant de la file */
  |   | Pour N ∈ fils(courant) Faire
  |   |   | OUVERTS ← enfiler(N, OUVERTS)
  |   | Finpour
  | Fintantque
Fin
```

FIGURE 2 – Parcours en largeur d'un arbre.

sur ses fils. L'algorithme de la figure 1 définit le parcours infixé d'un arbre. Si on applique cet algorithme sur l'arbre A ci-dessus avec un traitement consistant à afficher le nœud courant, le résultat est l'affichage $a\ b\ e\ f\ c\ g\ h\ d\ i$.

Un ***parcours en largeur*** d'un arbre consiste à traiter les nœuds par rang croissant : l'arbre lui-même, ses fils, ses petit-fils, etc. Sur l'exemple précédent, le résultat est l'affichage $a\ b\ c\ d\ e\ f\ g\ h\ i$. Un algorithme classique pour le faire s'appuie sur une file (ou file d'attente : une structure *FIFO — first in, first out —* dans laquelle les premiers éléments à quitter la file sont les plus anciennement entrés dans la file), file dont les éléments sont des arbres. Pour ce faire, on a besoin des opérations primitives suivantes :

- `file_vide` : $\longrightarrow$ `file` : constante du type des files (la file sans élément) ;
- `enfiler` : `arbre` $\times$ `file` $\longrightarrow$ `arbre` : la file `enfiler`(A, F) est la file obtenue en ajoutant (en queue de file) l'arbre A ;
- `est_vide` : `file` $\longrightarrow$ `booléen` permet de tester la vacuité d'une file ;
- `premier` : `file` $\longrightarrow$ `arbre` donne le premier élément de la file (celui qui a été introduit le plus tardivement) et déclenche une erreur si la file est vide ;
- `défiler` : `file` $\longrightarrow$ `file` : la file `défiler`(F) est obtenue en enlevant à F son premier élément et déclenche une erreur si la file est vide.

L'idée de l'algorithme est de créer une file avec l'arbre A (qu'on cherche à parcourir) comme seul élément. Puis, pour chaque premier élément de la file, on le traite, on l'enlève de la file et on ajoute à la file ses (éventuels) fils. Les nœuds qui sont dans cette file sont les nœuds dits *ouverts*, d'où le nom de cette file : `OUVERTS`. L'algorithme s'arrête quand la file est vide. L'algorithme de la figure 2 définit le parcours en largeur d'un arbre.

Le facteur de branchement d'un arbre est son nombre de fils. Le ***facteur de branchement moyen*** d'un arbre

est la moyenne arithmétique des facteurs de branchement des nœuds de l'arbre.

Si tous les nœuds d'un arbre en-dehors des nœuds-feuilles ont un facteur de branchement $b \geqslant 2$ et que la profondeur de cet arbre est p, alors cet arbre contient $b^0 + b^1 + \ldots + b^p = \dfrac{b^{p+1} - 1}{b - 1} = O(b^p)$.

En généralisant (sans prouver rigoureusement le résultat), on arrive à ceci : si on a un arbre quelconque et que $b > 1$ est son facteur de branchement moyen, le nombre de nœuds à la profondeur r est en $O(b^r)$.

Le chapitre suivant s'intéresse au parcours d'arbres potentiellement infinis. Ce parcours et la construction de nœuds de l'arbre coïncident : tous les nœuds ne sont pas en mémoire mais sont construits au fur et à mesure. Néanmoins, les notions et les principes algorithmiques présentés ci-dessus seront réutilisés.

*****

3 Parcours d'espaces d'états

3.1 Exemple introductif : le taquin

Le jeu du taquin est décrit comme suit. Un ***état*** est donné par un tableau 3×3 dont l'ensemble des valeurs est $\{1, 2, \ldots, 8, \blacksquare\}$. Par exemple, considérons l'état $\mathtt{e}_0 =$ $\begin{array}{|c|c|c|}\hline 4 & 1 & 3 \\ \hline \blacksquare & 2 & 7 \\ \hline 6 & 8 & 5 \\ \hline \end{array}$. La valeur $\blacksquare$ symbolise un « trou ». L'idée est de faire circuler le trou jusqu'à ce qu'on arrive à un état réalisant un but. Par exemple, on veut que la première ligne de l'état soit $\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline \end{array}$. On notera ce but par $\mathtt{but} =$ $\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline ? & ? & ? \\ \hline ? & ? & ? \\ \hline \end{array}$. Une action élémentaire pour faire circuler le trou est de l'échanger avec une case adjacente vers le haut (noté N, comme Nord), le bas (noté S, comme Sud), la droite (noté E, comme Est) ou la gauche (noté O, comme Ouest). Ainsi, à partir de l'état $\mathtt{e}_0$ ci-dessus, trois actions élémentaires sont possibles : l'action O ne l'est pas (le trou est déjà tout à gauche, il ne peut pas aller plus à gauche). On appelle ***successeurs*** d'un état e l'ensemble des états $\mathtt{successeurs(e)}$ qu'on peut obtenir en appliquant une action élémentaire sur e.

Un ***problème*** du taquin est donné par un état, appelé ***état initial***, un but et une fonction successeurs. Par exemple, $\mathtt{pb} = (\mathtt{e}_0, \mathtt{but}, \mathtt{successeurs})$ est un problème du taquin.

Une ***solution*** à un problème du taquin est une séquence d'états allant de son état initial à un état réalisant le but, sachant qu'un état non initial est successeur de son état précédent dans la séquence. Ainsi, une solution de pb est :

$$\mathtt{sol} = \begin{array}{|c|c|c|}\hline 4 & 1 & 3 \\ \hline \blacksquare & 2 & 7 \\ \hline 6 & 8 & 5 \\ \hline \end{array} \xrightarrow{\text{N}} \begin{array}{|c|c|c|}\hline \blacksquare & 1 & 3 \\ \hline 4 & 2 & 7 \\ \hline 6 & 8 & 5 \\ \hline \end{array} \xrightarrow{\text{E}} \begin{array}{|c|c|c|}\hline 1 & \blacksquare & 3 \\ \hline 4 & 2 & 7 \\ \hline 6 & 8 & 5 \\ \hline \end{array} \xrightarrow{\text{S}} \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & \blacksquare & 7 \\ \hline 6 & 8 & 5 \\ \hline \end{array}$$

L'***espace d'états*** du taquin est le graphe infini dont les sommets sont les états et les arcs sont les couples d'états $(\mathtt{e}, \mathtt{e}')$ tels que $\mathtt{e}' \in \mathtt{successeurs(e)}$. Ces arcs sont éventuellement étiquetés pour préciser l'action correspondant au passage de e vers $\mathtt{e}'$. Résoudre un problème du taquin revient dont à chercher un chemin sol dans l'espace d'états de l'état initial du problème à un état vérifiant le but.

L'***arbre de recherche généré par un état*** $\mathtt{e}_0$ est un arbre d'états A dont la racine est $\mathtt{e}_0$ et tel que, pour tout chaque nœud N de A et tout $F \in \mathtt{fils}(N)$ on ait $\mathtt{racine}(F) \in \mathtt{successeurs}(\mathtt{racine}(N))$. La figure 3 présente une partie de l'arbre de recherche pour l'état initial du problème précédent $\mathtt{e}_0$. On notera qu'un même état apparaîtra plusieurs fois dans l'arbre. En l'occurrence, l'espace d'états est fini (il comprend $9! = 362\,880$ états) alors que tout arbre de recherche du taquin est infini.

Question 1 *Pourquoi tout arbre de recherche du taquin est-il infini ?*

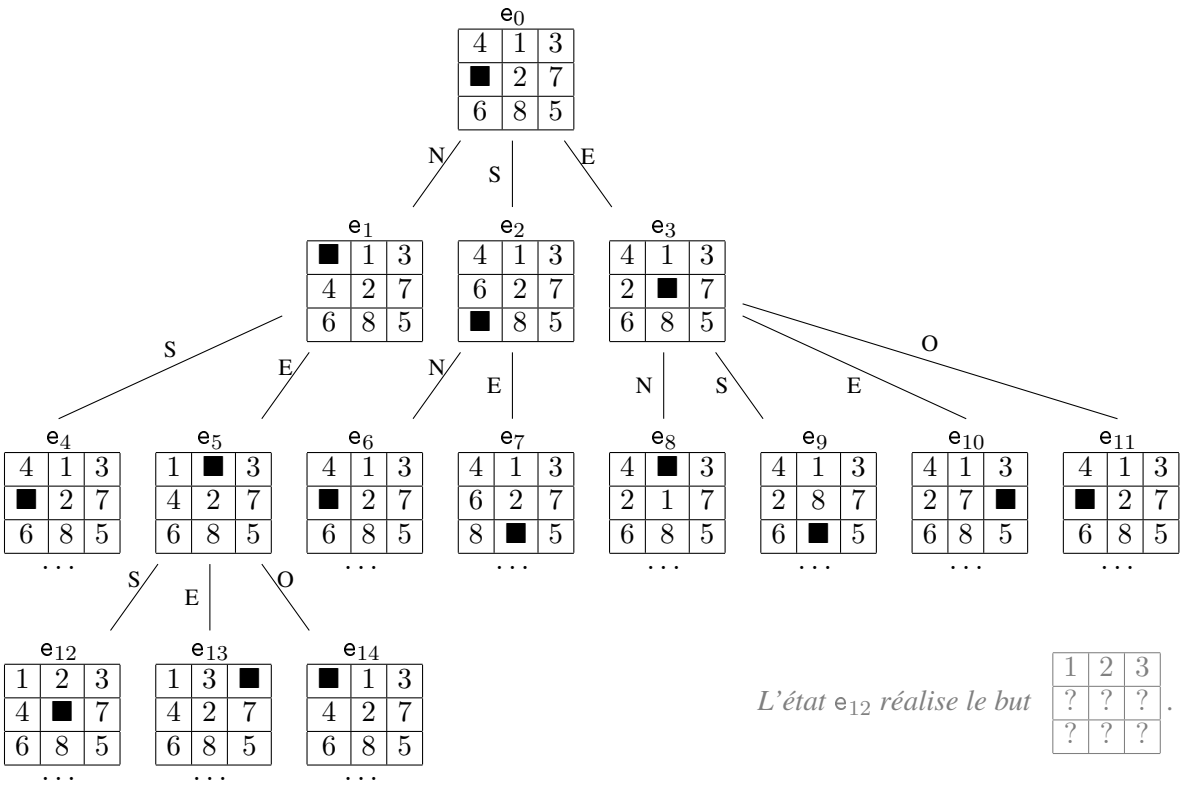


FIGURE 3 – Extrait d'un arbre de recherche pour le taquin.

3.2 Modélisation de problèmes discrets

De façon générale, on définit une classe de problèmes (telle que la classe des problèmes du taquin) de la façon suivante :

- On définit une notion d'***état*** (i.e., un type `état`).
- On définit une fonction `successeurs` qui à un état associe un ensemble fini d'états : le fait que l'état $\mathtt{e}'$ est obtenu par une « action élémentaire » à partir de l'état `e` se note donc $\mathtt{e}' \in \mathtt{successeurs}(\mathtt{e})$.
- On définit une notion de ***but***, un but étant une fonction `but` qui à un état associe un booléen : $\mathtt{but}(\mathtt{e}) = \mathtt{vrai}$ signifie que l'état `e` réaliste `but`.
- Un ***problème*** de cette classe est alors un triplet $(\mathtt{e}_0, \mathtt{but}, \mathtt{successeurs})$.

Concernant le but, deux cas peuvent se présenter :

- Soit la notion de but est la même pour tout problème et dans ce cas elle pourra être codée « en dur » (par une fonction ou une méthode du langage de programmation choisi) ;
- Soit le but varie d'un problème à un autre et dans ce cas, il faut représenter la notion de but.

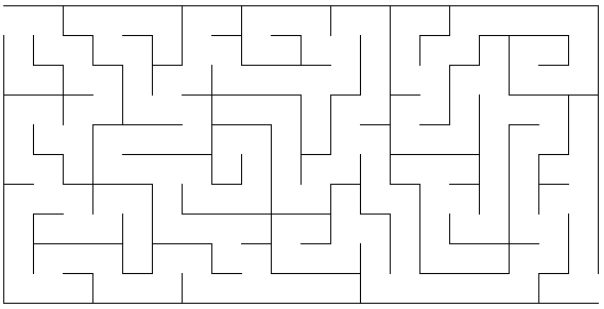
Une ***solution*** d'un problème $\mathtt{pb} = (\mathtt{e}_0, \mathtt{but}, \mathtt{successeurs})$ est une séquence d'états $\mathtt{e}_0 \longrightarrow \mathtt{e}_1 \longrightarrow \mathtt{e}_2 \longrightarrow \ldots \longrightarrow \mathtt{e}_f$ telle que :

- Le premier élément de la séquence est l'état initial ;
- Pour tout $i \in \{1, 2, \ldots n-1\}$, $\mathtt{e}_{i+1} \in \mathtt{successeurs}(\mathtt{e}_i)$;
- $\mathtt{but}(\mathtt{e}_f) = \mathtt{vrai}$ ($\mathtt{e}_f$ est un *état final*).

Exercice 1 *On considère les classes de problèmes suivants :*

I Problèmes de recherche de chemins dans un labyrinthe.

Le labyrinthe suivant a été généré à partir du site `http://www.desmoulins.fr/index.php?pg=divers!jeux!labyrinthes` :



II *Problèmes des n reines.* Soit $n \in \mathbb{N}$, $n \geqslant 1$. On considère un échiquier $n \times n$ et n reines de couleurs deux à deux différentes. Le problème est le suivant : comment poser ces reines pour qu'aucune ne menace directement l'autre [1] ?

III *Problème du cavalier.* Sur une case d'un échiquier classique (8×8), on pose un cavalier. Comment faire pour qu'il parcoure toutes les cases de l'échiquier une fois et une seule ?

IV *Les missionnaires et les cannibales.* Trois missionnaires et trois cannibales sont sur la rive d'un fleuve. Pour le traverser, il y a une barque à deux places, qu'on peut utiliser avec une ou deux personnes à bord. Or, dès que les cannibales sont en surnombre sur une des rives, ils mangent les missionnaires éventuels de cette rive. Comment organiser les traversées du fleuve pour que les six personnages aient traversé sans qu'aucun missionnaire ne soit mangé [2] ?

V *Jeu du cube de Rubik.*

VI *Jeu du solitaire.*

VII *Jeu du sudoku.*

VIII *Équation diophantienne* $2x - 3y + 6z = 17$. L'objectif est de trouver *une* solution de cette équation sur $\mathbb{Z}^3$.

IX *Isomorphisme de sous-graphe.* On a deux graphes G et H et on veut tester si G est un sous-graphe partiel de H à un isomorphisme près. La réponse positive à ce test signifie qu'il existe une fonction injective φ de l'ensemble des sommets de G dans l'ensemble des sommets de H telle que pour tout arc (x, y) de G, $(\varphi(x), \varphi(y))$ est un arc de H.

X *Problème SAT.* Étant donné une formule φ de la logique propositionnelle à n variables, on cherche à savoir s'il existe un modèle de φ.

Pour chacun de ces problèmes, il est demandé :

— *De le modéliser comme un problème de recherche dans un espace d'états ;*

— *D'étudier comment représenter de façon efficace un état (pour diminuer le temps de calcul et la place mémoire nécessaire).*

Certains ***problèmes de planification*** peuvent être considérés comme des problèmes de recherche dans des espaces d'états. C'est le cas en particulier pour les problèmes de planification dans des espaces discrets et pour lesquels il n'y a pas d'incertitude (l'univers et le résultat des actions sont connus). Pour définir un problème de planification, on définit des actions, une action étant représentée par des préconditions (quelles sont les conditions nécessaires et suffisantes pour qu'une action soit applicable ?) et des effets (souvent représentés par des connaissances à enlever et des connaissances à ajouter). Dans ce cadre, les successeurs d'un état $\mathtt{e}$ sont les états $\mathtt{e}'$ tels qu'il existe une action applicable sur $\mathtt{e}$ et dont l'application donne $\mathtt{e}'$.

Par exemple, le problème du loup, de la chèvre et du chou (qui est de la même famille que celui des missionnaires et des cannibales, cf. exercice 1, problème IV) peut se modéliser comme un problème de planification.

1. Dans ces notes de cours, les connaissances élémentaires sur le jeu d'échecs sont supposées connues, de même que les connaissances sur le cube de Rubik, le solitaire, le sudoku et autres jeux, dont il est facile de trouver une description en ligne.

2. Un quatrième cannibale sur la rive d'arrivée du fleuve était caché derrière un arbre. La suite de l'histoire est d'une grande tristesse.

Marcel doit faire traverser une rivière par un loup, une chèvre et un chou à l'aide de sa barque qui ne peut pas supporter plus d'un passager. Ils sont initialement sur la rive A et veulent atteindre la rive B. En l'absence du loup, la chèvre mange le chou et en absence du chou, le loup mange la chèvre. À la fin, tous doivent avoir traversé et aucun ne doit avoir été mangé. La côté initial de la rivière est le côté A, le côté final est le côté B. Pour modéliser ce problème, on considère les constantes `loup`, `chèvre`, `chou`, `rive_A`, `rive_B` et `barque`. L'état initial $\mathtt{e}_0$ est donné par la conjonction de littéraux suivants (en logique du premier ordre) :

$$\begin{aligned}\mathtt{e}_0 = {} & \mathtt{est\_sur\_la}(\mathtt{loup}, \mathtt{rive\_A}) \wedge \mathtt{est\_sur\_la}(\mathtt{chèvre}, \mathtt{rive\_A}) \wedge \mathtt{est\_sur\_la}(\mathtt{chou}, \mathtt{rive\_A}) \\ & \wedge \mathtt{barque\_sur\_rive\_A}()\end{aligned}$$

On dit qu'un état est interdit si la condition suivante est vérifiée :

$$\begin{aligned}\mathtt{interdit} = {} & \exists r \ (\mathtt{est\_sur\_la}(\mathtt{loup}, r) \wedge \mathtt{est\_sur\_la}(\mathtt{chèvre}, r) \wedge \neg \mathtt{est\_sur\_la}(\mathtt{chou}, r)) \\ & \vee (\neg \mathtt{est\_sur\_la}(\mathtt{loup}, r) \wedge \mathtt{est\_sur\_la}(\mathtt{chèvre}, r) \wedge \mathtt{est\_sur\_la}(\mathtt{chou}, r))\end{aligned}$$

De plus, on a les connaissances du domaine suivantes :

$$\mathtt{rive\_A} \neq \mathtt{rive\_B} \quad \wedge \quad \mathtt{rive\_A} \neq \mathtt{barque} \quad \wedge \quad \mathtt{rive\_B} \neq \mathtt{barque}$$
$$\forall x \forall r_1 \forall r_2 \ \mathtt{est\_sur\_la}(x, r_1) \wedge \mathtt{est\_sur\_la}(x, r_2) \Rightarrow r_1 = r_2$$

L'action `monter` prend en paramètre un des trois protagonistes et le fait monter dans la barque. Dans la modélisation discrète qui est faite, la barque est soit sur la rive A soit sur la rive B. Les conditions pour que l'action $\mathtt{monter}(x)$ soit possible est que la barque soit du même côté que x

$$\begin{aligned}&(\mathtt{est\_sur\_la}(x, \mathtt{rive\_A}) \wedge \mathtt{barque\_sur\_rive\_A}()) \vee \\ &(\mathtt{est\_sur\_la}(x, \mathtt{rive\_B}) \wedge \neg \mathtt{barque\_sur\_rive\_A}())\end{aligned}$$

que x soit l'un des trois protagonistes

$$x = \mathtt{loup} \vee x = \mathtt{chèvre} \vee x = \mathtt{chou}$$

et que le fait d'enlever x de la rive où il est ne conduit pas à un état interdit. L'effet de $\mathtt{monter}(x)$ sur l'état `e` sur lequel il est applicable conduit à la création d'un état $\mathtt{e}'$ obtenu en copiant `e`, en enlevant le littéral de la forme $\mathtt{est\_sur\_la}(x, r)$ et en ajoutant le littéral $\mathtt{est\_sur\_la}(x, \mathtt{barque})$.

On modélise de façon similaire l'action `descendre` (qui fait descendre de la barque).

Enfin, on modélise l'action `traverser` qui n'a pas de précondition (formalisé par une tautologie) et dont l'effet est :

— Si $\mathtt{barque\_sur\_rive\_A}()$, alors on enlève ce littéral et on ajoute le littéral $\neg\mathtt{barque\_sur\_rive\_A}()$;
— Sinon, d'enlever le littéral $\neg\mathtt{barque\_sur\_rive\_A}()$ et d'ajouter le littéral $\mathtt{barque\_sur\_rive\_A}()$.

3.3 Parcours en profondeur et en largeur d'un espace d'état

3.3.1 Principes généraux

Résoudre un problème de recherche $(\mathtt{e}_0, \mathtt{but}, \mathtt{successeurs})$ dans un espace d'états consiste donc à parcourir l'arbre d'états généré par $\mathtt{e}_0$ jusqu'à ce qu'un état réalisant `but` soit trouvé. On peut envisager de faire un parcours en profondeur ou en largeur, comme pour les arbres finis représentés explicitement (cf. section 2), mais en tenant compte du fait que ces arbres sont souvent infinis.

Il peut arriver également que le problème posé n'ait pas de solution (i.e., pour tout état `e` de l'arbre de recherches, $\mathtt{but}(\mathtt{e}) = \mathtt{faux}$). Dans ce cas :

— Soit l'arbre de recherche est infini, auquel cas, la recherche ne terminera pas ;
— Soit l'arbre de recherche est fini, auquel cas, la recherche se terminera par un échec (`échec` est une constante du type des solutions).

Quand un tel arbre de recherche est infini, une recherche en profondeur peut échouer, même s'il existe une solution, alors qu'une recherche en largeur réussira toujours (pour peu qu'on ait assez de temps et d'espace mémoire).

Fonction `recherche_profondeur` $((\mathtt{e}_0, \mathtt{but}, \mathtt{successeurs})$: problème de recherche$)$: `solution`
Variables
  s : `état`
  `sol` : `solution`
Début
  Si $\mathtt{but}(\mathtt{e}_0)$ **Alors**
    retourner $\mathtt{e}_0$
  Finsi
  Pour $s \in \mathtt{successeurs}(\mathtt{e}_0)$ **Faire**
    $\mathtt{sol} \leftarrow \mathtt{recherche\_profondeur}((s, \mathtt{but}, \mathtt{successeurs}))$
    Si $\mathtt{sol} \neq \mathtt{échec}$ **Alors**
      retourner $\mathtt{e}_0 \longrightarrow \mathtt{sol}$
      /* *Si* $\mathtt{sol} = \mathtt{e}_1 \longrightarrow \ldots \longrightarrow \mathtt{e}_f$ *alors* $\mathtt{e}_0 \longrightarrow \mathtt{sol} = \mathtt{e}_0 \longrightarrow \mathtt{e}_1 \longrightarrow \ldots \longrightarrow \mathtt{e}_f$ */
    Finsi
  Finpour
retourner `échec`
Fin

FIGURE 4 – Recherche en profondeur dans un espace d'états.

3.3.2 Algorithme de recherche en profondeur

L'algorithme de la figure 4 présente la ***recherche en profondeur*** dans un espace d'états. Il est à comparer avec l'algorithme de parcours en profondeur d'un arbre fini (cf. figure 1). Le résultat est une valeur de type `solution`, c'est-à-dire soit une séquence (une liste) d'états, qu'on notera $\mathtt{e}_0 \longrightarrow \mathtt{e}_1 \longrightarrow \mathtt{e}_2 \longrightarrow \ldots \longrightarrow \mathtt{e}_f$, soit la constante `échec`.

Notons que pour certains problèmes, seul l'état final $\mathtt{e}_f$ de la solution est intéressant. C'est le cas par exemple pour le problème des n reines (cf. exercice 1, problème II).

À noter que la méthode du ***backtracking*** (algorithme avec retours en arrière ?) en algorithmique, peut être considéré comme une recherche en profondeur d'abord.

3.3.3 Algorithme de recherche en largeur

L'algorithme de la figure 5 présente la ***recherche en largeur*** dans un espace d'états. Il est à comparer avec l'algorithme de parcours en largeur d'un arbre fini (cf. figure 2). La dernière instruction fait appel à la fonction `état_final_en_solution` qui mérite une explication. Comme il est plus simple de manipuler des états que des séquences d'états, l'algorithme cherche un état final $\mathtt{e}_f$. On suppose qu'un pointeur est mis en place entre un état généré par la fonction `successeurs` et l'argument de cette fonction, de façon à récupérer l'antécédent d'un état. De cette façon, on peut obtenir la séquence $\mathtt{e}_f \longrightarrow \ldots \longrightarrow \mathtt{e}_1 \longrightarrow \mathtt{e}_0$ qu'il ne reste plus qu'à inverser et le résultat de cette inversion est retourné par la fonction `état_final_en_solution`.

Question 2 *Quels sont les critères pour préférer une recherche en profondeur à une recherche en largeur ou l'inverse ?*

Question 3 *Que peut-on dire sur la longueur d'une solution produite par une recherche en largeur ? Et pour la longueur d'une solution produite par une recherche en profondeur ?*

Exercice 2 *On considère le problème du taquin et les problèmes donnés dans l'exercice 1. Pour chacun de ces exercices, discutez du choix entre une recherche en profondeur et une recherche en largeur.*

```
Fonction recherche_largeur ((e_0, but, successeurs) : problème de recherche) : solution
Variables
    OUVERTS : file d'états
    courant, e_f : état
Début
    OUVERTS ← enfiler(e_0, file_vide)
    Tant que non est_vide(OUVERTS) et non but(premier(OUVERTS)) Faire
        courant ← premier(OUVERTS)
        OUVERTS ← défiler(OUVERTS)        /* enlever courant de la file */
        Pour s ∈ successeurs(courant) Faire
            OUVERTS ← enfiler(s, OUVERTS)
        Finpour
    Fintantque
    Si est_vide(OUVERTS) Alors
        retourner échec
    Finsi
    e_f ← premier(OUVERTS)
    retourner état_final_en_solution(e_f)
Fin
```

FIGURE 5 – Recherche en largeur dans un espace d'états.

3.3.4 À propos de la complexité

La recherche en profondeur peut ne jamais aboutir, même si une solution existe.

La recherche en largeur a une complexité en $O(b^p)$ où b est le facteur de branchement moyen et p est la longueur de la première solution rencontrée (donc la longueur de la plus courte solution). On notera que cette complexité est à la fois celle du pire cas et celle du meilleur cas (le rapport entre le temps de calcul dans le pire cas et le temps de calcul dans le meilleur cas est en $O(1)$).

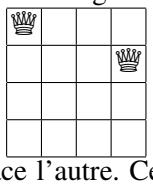
Exercice 3 *On considère le problème du taquin et les problèmes donnés dans l'exercice 1.*
Pour chacun de ces problèmes, que peut-on dire sur b et sur p ?

3.3.5 Variantes et optimisations

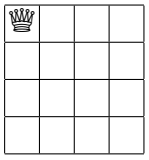
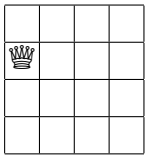
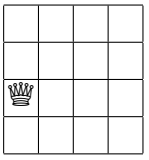
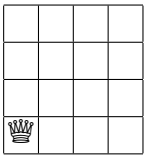
Recherche en profondeur limitée. Supposons qu'on ait un arbre de recherche infini, mais qu'on ait peu de temps pour implanter l'algorithme de recherche. La recherche en profondeur étant plus simple à implanter que la recherche en largeur (qui suppose l'implantation d'un type pour les files d'états), on peut penser à un algorithme de ***recherche en profondeur limitée*** : on limite l'arbre de recherche à une profondeur fixée `prof_max`.

Cet algorithme termine toujours et sa complexité dans le pire cas est en $O(b^{\texttt{prof\_max}})$.

Éviter de générer plusieurs fois le même état. On a vu, dans l'exemple du taquin, que l'arbre de recherche peut contenir plusieurs fois le même état. Cela induit une complexité supplémentaire puisque cela augmente le facteur de branchement moyen et, pour la recherche en profondeur, cela peut augmenter la profondeur de l'arbre parcouru. La solution est donc d'éviter de générer des états qui ont déjà été générés, en gérant l'ensemble `EÉDG` des états déjà générés. Ainsi, `EÉDG` est initié avec $\{\mathtt{e}_0\}$, au lieu de `successeurs`(x) on écrit `successeurs`$(x) \setminus$ `EÉDG` et à chaque appel à `successeurs`(x) on ajoute les états générés à `EÉDG`.

Réduire l'ensemble des états parcourus. Dans certaines applications, on peut ne considérer qu'un sous-ensemble de tous les états envisageables, si cela ne nuit pas à la résolution du problème. Dans l'exemple des 4 reines, par exemple, l'état  peut être envisagé : il contient $k = 2$ reines, avec $k \leqslant n = 4$ et aucune des deux reines ne menace l'autre. Cependant, on peut interdire ce genre d'états en considérant que les k reines sont sur les k premières colonnes. Cela n'enlève aucune solution : toutes les solutions peuvent être générées en 4 étapes, partant de l'état initial et en ne considérant que des états respectant cette nouvelle contrainte. L'avantage de cela est de limiter la complexité en baissant le facteur de branchement. Ainsi, dans le problème des n reines dont l'état initial est l'échiquier $n \times n$ vide, on peut considérer en théorie n^2 successeurs de cet état initial, alors qu'avec cette contrainte supplémentaire, on n'en considère que n.

Une autre façon de réduire la complexité consiste à tenir compte de *symétries*. En reprendant l'exemple des 4 reines, avec la contrainte introduite ci-dessus, il y a quatre successeurs à l'état initial :

$\mathtt{e}_1 =$  $\mathtt{e}_2 =$  $\mathtt{e}_3 =$  $\mathtt{e}_4 =$ 

Si on considère la symétrie σ d'axe horizontal de l'échiquier, on constate que $\mathtt{e}_3 = \sigma(\mathtt{e}_1)$ et $\mathtt{e}_4 = \sigma(\mathtt{e}_2)$. Par conséquent si on a une solution dont le deuxième état est $\mathtt{e}_1$ ou $\mathtt{e}_2$ alors on peut trouver une solution à partir de l'état $\mathtt{e}_3$ ou de l'état $\mathtt{e}_4$ en utilisant la symétrie, et réciproquement. Donc, si l'objectif est de trouver une solution, on peut très bien ne pas considérer les états $\mathtt{e}_3$ et $\mathtt{e}_4$, ce qui fait qu'il n'y aura que deux successeurs à l'état initial et que l'arbre de recherche sera diminué (de la moitié), d'où un gain en temps de calcul.

La notion de symétrie est très dépendante d'une application. Par exemple, quand on parle de symétrie de graphes, il s'agit de leurs automorphismes.

Exercice 4 *Écrivez un algorithme récursif pour la recherche en profondeur limitée. La signature de la fonction doit être :*

Fonction `recherche_profondeur_limitée` $((\mathtt{e}_0, \mathtt{but}, \mathtt{successeurs})$: problème de recherche, `prof_max` : `entier naturel`$)$: `solution`

Exercice 5 *Au lieu de chercher* une *solution, on peut vouloir générer* toutes *les solutions d'un problème de recherche et les afficher au fur et à mesure (le processus ne terminera alors pas quand l'arbre de recherche est infini).*

Comment modifier l'algorithme de recherche en largeur pour cela ?

Comment modifier l'algorithme de recherche en profondeur pour cela, sous l'hypothèse d'un arbre de recherche fini ?

3.4 L'algorithme A*

3.4.1 Principes

Considérons à nouveau l'arbre de recherche du taquin tel qu'il est décrit à la figure 3. La recherche en largeur d'abord consiste à parcourir les états $\mathtt{e}$ à $\mathcal{F}(\mathtt{e})$ croissant où $\mathcal{F}(\mathtt{e})$ calcule le rang de l'état $\mathtt{e}$ dans l'arbre de recherche (0 pour l'état initial, 1 pour les successeurs de l'état initial, 2 pour les successeurs des successeurs de l'état initial, etc.). Pour deux nœuds ayant le même rang, l'ordre est arbitraire.

Pourrait-on trouver une autre fonction $\mathcal{F}$ qui permette d'aboutir plus rapidement à une solution ? Une réponse à cette question consiste à prendre la fonction $\mathcal{F}^*$ définie ainsi :

$$\mathcal{F}^*(\mathtt{e}) = \text{longueur de la plus courte solution passant par } \mathtt{e} \text{ et allant de } \mathtt{e}_0 \text{ à un état réalisant le but}$$

Soit $f = \mathcal{F}^*(\mathtt{e}_0)$. Pour trouver une solution au problème $(\mathtt{e}_0, \mathtt{but}, \mathtt{successeurs})$ en utilisant $\mathcal{F}^*$, il suffit de choisir un successeur $\mathtt{e}_1$ de $\mathtt{e}_0$ tel que $\mathcal{F}^*(\mathtt{e}_1) = \mathcal{F}^*(\mathtt{e}_0)$, puis un successeur $\mathtt{e}_2$ de $\mathtt{e}_1$ tel que $\mathcal{F}^*(\mathtt{e}_2) = \mathcal{F}^*(\mathtt{e}_1)$, etc. Le f^{e} état ainsi généré, $\mathtt{e}_f$, réalisera nécessairement le but et on aura une solution $\mathtt{e}_0 \longrightarrow \mathtt{e}_1 \longrightarrow \ldots \longrightarrow \mathtt{e}_f$. De plus, cette solution sera ***optimale*** : c'est une solution de longueur minimale au problème de recherche.

Malheureusement, dans beaucoup d'applications, aucune implantation efficace de $\mathcal{F}^*$ n'est connue. L'idée est alors de trouver une façon d'approcher $\mathcal{F}^*$. Pour cela, on décompose $\mathcal{F}^*$ en une somme :

$$\begin{aligned} \mathcal{F}^*(\mathtt{e}) &= \mathcal{G}^*(\mathtt{e}) + \mathcal{H}^*(\mathtt{e}) \\ \text{avec } \mathcal{G}^*(\mathtt{e}) &= \text{longueur du plus court chemin allant de } \mathtt{e}_0 \text{ à un } \mathtt{e} \\ \text{et } \mathcal{H}^*(\mathtt{e}) &= \text{longueur du plus court chemin allant de } \mathtt{e} \text{ à un état réalisant le but} \end{aligned}$$

et on approche $\mathcal{G}^*$ par $\mathcal{G}$ et $\mathcal{H}^*$ par $\mathcal{H}$, d'où la fonction $\mathcal{F}$ définie par $\mathcal{F}(e) = \mathcal{G}(e) + \mathcal{H}(e)$ qui approche $\mathcal{F}^*$.

On dit que la fonction $\mathcal{F}$ est ***admissible*** si, pour tout état $\mathtt{e}$,

$$\begin{aligned} \mathcal{G}(\mathtt{e}) &\geqslant \mathcal{G}^*(\mathtt{e}) && (\mathcal{G} \text{ est admissible}) \\ \text{et } \mathcal{H}(\mathtt{e}) &\leqslant \mathcal{H}^*(\mathtt{e}) && (\mathcal{H} \text{ est admissible}) \end{aligned}$$

On peut montrer que si la fonction $\mathcal{F}$ est admissible alors une solution obtenue en parcourant l'arbre de recherche par $\mathcal{F}(e)$ croissante est optimale.

3.4.2 Choix de $\mathcal{G}$

Le plus simple pour $\mathcal{G}(\mathtt{e})$ est de prendre le nombre d'étapes qui ont effectivement mené à l'état $\mathtt{e}$. On a $\mathcal{G}(\mathtt{e}) \geqslant \mathcal{G}^*(\mathtt{e})$ puisque $\mathcal{G}^*(\mathtt{e})$ est le minimum d'un ensemble de longueurs de chemins dont $\mathcal{G}(\mathtt{e})$ est un élément.

3.4.3 Choix de $\mathcal{H}$ (l'heuristique)

La fonction $\mathcal{H}$ s'appelle l'***heuristique***. Son choix dépend largement de la classe de problèmes qu'on est en train de traiter. Une fonction heuristique admissible est la fonction constamment nulle. Cependant, plus proche la valeur de $\mathcal{H}(\mathtt{e})$ est de $\mathcal{H}^*(\mathtt{e})$ (tout en restant inférieure ou égale), plus courts seront les temps de calcul.

Un principe général qui guide le choix de l'heuristique est le suivant :

> Si on cherche un minorant m de la longueur d'un plus court chemin dans un espace contenant des « obstacles », il suffit de chercher un plus court chemin négligeant ces obstacles.

Par exemple, si on veut une valeur par défaut de la distance entre Nancy et Toulouse en voiture, on peut prendre la distance « à vol d'oiseau » entre ces deux villes (distance qui ne tient pas compte du fait que les voitures ont la contrainte de rouler sur des routes).

Comme $\mathcal{H}^*(\mathtt{e})$ est la longueur d'un chemin de $\mathtt{e}$ à un état final il suffit de trouver un chemin de $\mathtt{e}$ à un état final qui néglige certaines contraintes.

Appliquons ce principe au problème du taquin, sur l'état $\texttt{e} = \texttt{e}_0$ et le but de la figure 3 :

$$\texttt{e} = \begin{array}{|c|c|c|}\hline 4 & 1 & 3 \\ \hline \blacksquare & 2 & 7 \\ \hline 6 & 8 & 5 \\ \hline \end{array} \qquad \texttt{but} = \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline ? & ? & ? \\ \hline ? & ? & ? \\ \hline \end{array}$$

La contrainte que nous allons négliger est la suivante :

On ne peut déplacer une case avec un chiffre que vers une case avec un trou.

Si on néglige cette contrainte, on peut déplacer vers le haut la case avec le 2 : sur cette case, il y aura à la fois 1 et 2. Puis, on peut déplacer le 1 de cette case vers la gauche et on obtiendra la case en haut à gauche avec à la fois 1 et 4. L'état obtenu satisfait le but (même si ce n'est pas un état au sens défini à l'origine). En résumé, les déplacements dans un espace d'états avec contrainte relâchée sont les suivants :

$$\begin{array}{|c|c|c|}\hline 4 & 1 & 3 \\ \hline \blacksquare & 2 & 7 \\ \hline 6 & 8 & 5 \\ \hline \end{array} \xrightarrow{\text{2N}} \begin{array}{|c|c|c|}\hline 4 & 1/2 & 3 \\ \hline \blacksquare & \blacksquare & 7 \\ \hline 6 & 8 & 5 \\ \hline \end{array} \xrightarrow{\text{1O}} \begin{array}{|c|c|c|}\hline 1/4 & 2 & 3 \\ \hline \blacksquare & \blacksquare & 7 \\ \hline 6 & 8 & 5 \\ \hline \end{array}$$

Or, dans l'espace étendu des états (celui dans lequel il peut y avoir plusieurs valeurs par case et plusieurs trous), ce chemin est un chemin optimal pour aller de e à un état satisfaisant le but : il faut au moins faire une action pour que le 1 soit à la bonne position et au moins une action pour que le 2 soit à la bonne position. De cette façon, on arrive à $\mathcal{H}(\texttt{e}) = 2$ (le nombre d'étapes).

L'heuristique admissible possible pour le problème du taquin qui se généralise de l'exemple précédent est définie par

$$\mathcal{H}(e) = \sum_{x \text{ : entier ou } \blacksquare \text{ apparaissant dans but } (i \neq ?)} \begin{array}{c}\text{distance de Manhattan entre la case de } e \text{ contenant } x \\ \text{et la case de but contenant } x\end{array}$$

Notons que cette heuristique dépend également de but, mais comme le but ne varie pas, on n'a pas coutume de le mettre en paramètre. On rappelle que la distance de Manhattan entre deux points (i_1, j_1) et (i_2, j_2) est $|i_2 - i_1| + |j_2 - j_1|$.

Question 4 *On aurait pu relâcher davantage les contraintes et dire en plus que dans le nouvel espace d'états, une valeur pouvait se déplacer en un coup de n'importe quelle case à n'importe quelle autre case. Cela aurait permis de définir une autre heuristique $\mathcal{H}_2$. Laquelle ? Parmi $\mathcal{H}$ et $\mathcal{H}_2$, laquelle est la meilleure ? En quoi est-elle meilleure ?*

Exercice 6 *On considère le problème du labyrinthe (cf. exercice 1, problème I).*
Proposez deux heuristiques admissibles non nulles pour ce problème dont une est meilleure que l'autre.

Exercice 7 *On considère le problème des missionnaires et des cannibales (cf. exercice 1, problème IV).*
Proposez une heuristique admissible pour ce problème.

Exercice 8 *On considère le problème des n reines (cf. exercice 1, problème II). Pourquoi l'approche A* ne permet pas d'améliorer la recherche par rapport à un parcours en largeur ?*
Donnez un autre problème de l'exercice 1 qui est dans cette situation.

```
Fonction recherche_A* ((e_0, but, successeurs) : problème de recherche) : solution
Variables
    OUVERTS : liste d'états
    courant, e_f : état
Début
    OUVERTS ← cons(e_0, l_vide)
    Tant que non est_vide(OUVERTS) et non but(prem(OUVERTS)) Faire
        courant ← prem(OUVERTS)
        OUVERTS ← reste(OUVERTS)        /* enlever courant de la file */
        Pour s ∈ successeurs(courant) Faire
            Insérer s dans la liste OUVERTS de façon à ce que cette liste soit toujours à F(e) croissant
        Finpour
    Fintantque
    Si est_vide(OUVERTS) Alors
        retourner échec
    Finsi
    e_f ← prem(OUVERTS)
    retourner état_final_en_solution(e_f)
Fin
```

FIGURE 6 – Algorithme A*.

3.4.4 Algorithme

L'algorithme A* est donné à la figure 6. On voit qu'il s'inspire de l'algorithme de recherche en largeur (figure 5). La différence principale vient de la gestion de la liste des états ouverts : `OUVERTS` est une liste d'états triée par $\mathcal{F}(e)$ croissants (on parle parfois de file d'attente prioritaire). On rappelle les opérations primitives sur les listes :

- `l_vide` est la liste vide ;
- $\texttt{cons}(x, L)$ est la liste obtenue en ajoutant l'élément x en tête de L (par exemple, $(\texttt{e}_3\ \ \texttt{e}_2\ \ \texttt{e}_4) = \texttt{cons}(\texttt{e}_3, \texttt{cons}(\texttt{e}_2, \texttt{cons}(\texttt{e}_4, \texttt{l\_vide})))$) ;
- $\texttt{est\_vide}(L)$ donne `vrai` si L est vide et `faux` sinon ;
- $\texttt{prem}(L)$ donne le premier élément de la liste L ;
- $\texttt{reste}(L)$ donne la liste amputée de son premier élément.

Exercice 9 *Résolvez par A* le problème du taquin donné en section 3.1 en utilisant l'heuristique $\mathcal{H}^*$ de la section 3.4.3.*

3.5 Prise en compte du coût d'une transition

Dans ce qui précède, toutes les actions élémentaires permettant d'aller d'un état $\texttt{e}$ à un état $\texttt{e}' \in \texttt{successeurs}(\texttt{e})$ sont pondérées de la même façon. Par exemple, pour le taquin, on n'a pas différencié les 4 actions possibles dans un état donné.

Cette hypothèse (jusqu'alors implicite) n'est pas réaliste pour toutes les applications. Par exemple, le problème de la recherche d'un plus court chemin entre deux villes sur une carte, sachant qu'une action élémentaire consiste à se déplacer d'une ville à une ville voisine. Ce problème particulier se modélise par un problème de recherche de

chemin minimal dans un graphe valué : les sommets représentent les villes (considérées comme ponctuelles, pour simplifier) et les arcs, les tronçons de routes entre villes [3].

Étant donné une transition d'un état $\mathtt{e}$ à un état $\mathtt{e}'$, notée $\mathtt{e} \longrightarrow \mathtt{e}'$ (ou $\mathtt{e} \xrightarrow{\ell} \mathtt{e}'$ où ℓ est une étiquette expliquant le type de la transformation : $\ell \in \{\mathrm{N}, \mathrm{S}, \mathrm{E}, \mathrm{O}\}$ pour le taquin), son coût est noté $\mathtt{coût}\,(\mathtt{e} \longrightarrow \mathtt{e}')$ (ou $\mathtt{coût}\left(\mathtt{e} \xrightarrow{\ell} \mathtt{e}'\right)$).

Pour donner un exemple, si on suppose qu'un tableau du taquin est posé contre un mur et que déplacer une pièce vers le haut demande beaucoup d'effort (les pièces étant lourdes), que déplacer une pièce vers le bas demande peu d'effort et que les déplacements horizontaux demandent un effort moyen, on pourrait pondérer les transitions ainsi : $\mathtt{coût}\left(\mathtt{e} \xrightarrow{\mathrm{N}} \mathtt{e}'\right) = 1$, $\mathtt{coût}\left(\mathtt{e} \xrightarrow{\mathrm{S}} \mathtt{e}'\right) = 4$, $\mathtt{coût}\left(\mathtt{e} \xrightarrow{\mathrm{E}} \mathtt{e}'\right) = \mathtt{coût}\left(\mathtt{e} \xrightarrow{\mathrm{O}} \mathtt{e}'\right) = 2$ (faire déplacer une pièce vers le haut est la transition S, qui consiste à faire monter la pièce, le trou allant vers le bas).

Pour changer l'algorithme, la solution consiste juste à changer la fonction $\mathcal{F}$ en changeant $\mathcal{G}$ et $\mathcal{H}$. Pour $\mathcal{G}$, au lieu de compter le nombre d'étapes menant de l'état initial à l'état courant, on fait la somme des coûts ayant mené à l'état courant.

Exercice 10 *Proposez et discutez une fonction $\mathcal{H}$ pour le taquin avec prise en compte des coûts.*

Remarque 1 Dans le cas où la fonction $\mathcal{H}$ est constamment nulle, l'algorithme A* avec prise en compte des coûts des transitions devient très similaire à un l'algorithme dit de programmation dynamique, qui relève de la recherche opérationnelle.

*****

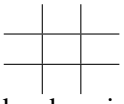
4 Les jeux à deux joueurs à somme nulle et information complète

Dans cette section sont abordés les jeux à deux joueurs. L'idée est d'écrire un programme qui puisse affronter un adversaire (humain, machine ou autre). Les jeux abordés sont les jeux dans lesquels il n'y a pas de hasard (sauf éventuellement pour déterminer lequel des deux joueurs commence) et tels que les deux joueurs ont une information complète de l'ensemble du jeu (pas de carte que ne verrait qu'un joueur, etc.). De plus, ce sont des jeux dits à somme nulle, ce qui signifie en particulier que l'objectif est de gagner et qu'il n'y a qu'un gagnant.

On peut citer les exemples de jeux suivants : les dames, les échecs [4], le go, Othello, etc.

Nous commencerons par un jeu simple puis aborderons les jeux plus complexes (en terme de complexité algorithmique).

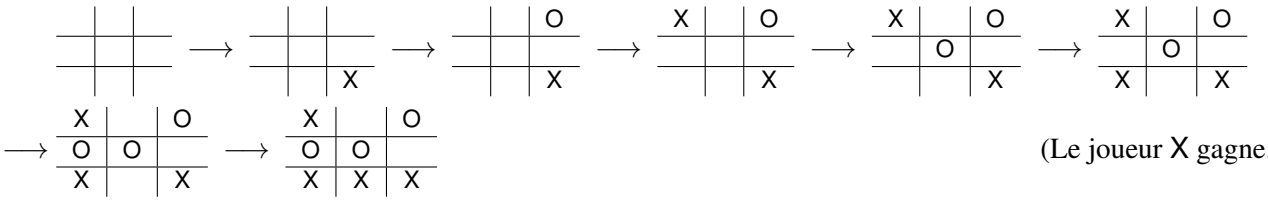
4.1 Un premier exemple : le morpion 3×3

Règles du jeu. Deux joueurs, appelés X et O remplissent une grille 3×3 initialement vide —  — en mettant tour à tour leurs symboles (X et O) dans une case encore vide. Le jeu s'arrête quand un des deux joueurs peut aligner trois cases (horizontalement, verticalement ou selon une des deux diagonales) ou quand la grille est pleine.

3. Le problème de recherche d'un chemin optimal dans un graphe fini est donc un problème de recherche dans un espace d'états. Néanmoins, des techniques particulières pour résoudre ce problème, plus efficaces qu'une approche telle que A*, existent (et sont classiquement enseignées dans des cours de recherche opérationnelle).

4. Pour les échecs, le fait qu'il y ait possibilité d'égalité — le pat — fait qu'il devrait être considéré à part, mais nous négligerons cet aspect.

Par convention, on supposera que le joueur X commence toujours. Voici le déroulement d'une partie :



(Le joueur X gagne.)

De la même façon qu'on avait un arbre de recherche dans le chapitre précédent, on peut créer un arbre dont chaque nœud contient un état de la partie, dont la racine est l'état initial et dont les fils d'un nœud sont obtenus par les coups qui peuvent être joués dans cet état. Un tel arbre contient moins de

$$1 + 9 + 9 \times 8 + 9 \times 8 \times 7 + \ldots 9 \times 8 \times 7 \times 6 + \ldots + 9 \times 8 \times \ldots \times 1 = 986\,410 \text{ états}$$

C'est un majorant, car dans certaines situations, un des joueurs peut gagner avant que la grille soit remplie [5].

Ce nombre est important pour un humain mais facile à gérer avec une machine actuelle. On peut donc générer l'arbre en entier. L'ensemble des parties possibles correspondra à l'ensemble des branches de l'arbre.

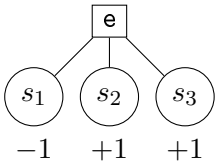
Cet arbre peut être utilisé pour écrire un programme qui joue au morpion : les adversaires seront appelés le joueur-machine et le joueur humain. Une fois cet arbre généré, on peut partitionner les états finaux $\texttt{e}_f$ (aux feuilles de l'arbre) en trois et leur associer une valeur $\texttt{éval}(\texttt{e}_f)$ de la façon suivante :

— État perdant (le joueur-machine a perdu) : $\texttt{éval}(\texttt{e}_f) = -1$;
— État gagnant (le joueur-machine a gagné) : $\texttt{éval}(\texttt{e}_f) = +1$;
— État nul (égalité) : $\texttt{éval}(\texttt{e}_f) = 0$.

Une action conduisant à un état négatif (i.e., un état e tel que $\texttt{éval}(\texttt{e}) < 0$) est à éviter par le joueur-machine et à privilégier par le joueur humain. Une action conduisant à un état positif (i.e., un état e tel que $\texttt{éval}(\texttt{e}) > 0$) est à privilégier par le joueur-machine et à éviter par le joueur humain. Pour faire un choix d'action à partir d'un état e, il serait utile de connaître les valeurs $\texttt{éval}(s)$ pour tout $s \in \texttt{successeurs}(\texttt{e})$ où $\texttt{successeurs}(\texttt{e})$ est l'ensemble des états qu'on peut atteindre en un coup à partir de l'état e : on choisira un coup menant à un état s maximisant $\texttt{éval}(s)$. Pour ce faire, il faut étendre la fonction éval, qui n'est pour l'instant définie que pour les états finaux, l'idée étant que plus $\texttt{éval}(\texttt{e})$ est grand, plus la situation est favorable.

La valeur de $\texttt{éval}(\texttt{e})$ sera calculée à partir des valeurs des $\texttt{éval}(s)$ pour $s \in \texttt{successeurs}$. en distinguant deux cas :

Premier cas : e est l'état à partir duquel le joueur-machine doit jouer. Si e a trois successeurs dont deux sont évalués à $+1$ et un à -1, la partie de l'arbre contenant e et ses successeurs se présente ainsi :



Par convention graphique, les états dans lequel le joueur-machine va jouer sont encadrés alors que les états dans lequel le joueur humain va jouer son encerclés.

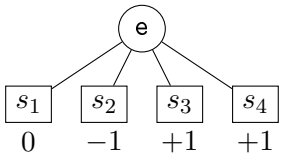
Dans cet exemple, s_1 est une situation moins favorable que s_2 ou s_3. Comme on veut programmer le joueur-machine afin qu'il gagne, il devra choisir s_2 ou s_3 plutôt que s_1. Du coup, l'évaluation associée à e sera $\max\{\texttt{éval}(s_1), \texttt{éval}(s_2), \texttt{éval}(s_3)\} = +1$.

De façon générale :

Si e est un état dans lequel la machine va jouer, $\texttt{éval}(\texttt{e}) = \max\{\texttt{éval}(s) \mid s \in \texttt{successeurs}(\texttt{e})\}$ (1)

5. On peut diminuer ce nombre en tenant compte de symétries : il y a 9 premiers coups possibles, mais seulement 3 coups réellement différents (au centre, dans un coin ou au milieu d'un bord) : les autres se déduisent par symétrie. On peut réduire le nombre de deuxième coup à un nombre dépendant de la nature du premier coup, mais variant entre 2 (si le premier coup est au centre) et 6 (si le premier coup est à un coin). En tenant compte de ces symétries, on peut donc majorer le nombre de nœuds de l'état obtenu en remplaçant les occurrences de 9 par des 3 et les occurrences des 8 par des 6 dans l'expression ci-dessus, ce qui donne moins de 328 805 états.

Deuxième cas : e est l'état à partir duquel le joueur humain doit jouer. Dans un exemple à 4 successeurs dont les valeurs sont évaluées à 0, -1, $+1$ et $+1$, on aura le schéma suivant :



La pire situation pour le joueur-machine est quand le joueur humain s'arrange pour choisir la situation menant à l'évaluation la plus basse. Du coup, l'évaluation associée à e sera $\min\{\texttt{éval}(s_1), \texttt{éval}(s_2), \texttt{éval}(s_3), \texttt{éval}(s_4)\} = -1$.

De façon générale :

Si e est un état dans lequel l'humain va jouer, $\texttt{éval}(\texttt{e}) = \min\{\texttt{éval}(s) \mid s \in \texttt{successeurs}(\texttt{e})\}$ (2)

On rappelle que si $A \subseteq \mathbb{R}$, $A \neq \emptyset$ et A majoré alors $\max_{x\in A} x = -\min_{x\in A} -x$. Par conséquent, si on considère que pour le joueur humain, la fonction d'évaluation est $-\texttt{éval}$, alors l'équation (2) correspond à l'équation (1) prise du point de vue du joueur humain. De même, l'équation (1) correspond à l'équation (2) prise du point de vue du joueur humain. Entre les deux joueurs, la situation est symétrique : ce qui est favorable à l'un est défavorable à l'autre.

4.2 L'algorithme minimax

Théoriquement, l'approche précédente s'applique pour bien d'autres jeux, comme le jeu d'Othello (ou de Réversi : les deux sont très proche). Cependant, une estimation (grossière) de ce jeu donne $3^{8^2} \simeq 3{,}4 \cdot 10^{30}$ états, ce qui rend cette approche rédhibitoire en espace mémoire et en temps de calcul (on estime qu'il y a environ 10^{27} atomes dans le corps humain et que le temps depuis la création de l'univers selon la théorie du Big Bang est de l'ordre de $4{,}36 \cdot 10^{26}$ nanosecondes).

L'idée est alors de ne parcourir l'arbre que jusqu'à un rang donné qui correspond au nombre de coups futurs qui sont explorés à partir de l'état courant. La difficulté est alors de définir une fonction d'évaluation pour les feuilles de l'arbre tronqué : celles-ci ne sont pas nécessairement des états finaux. On notera cette fonction d'évaluation $\texttt{éval}_0$: c'est l'***évaluation d'un état en regardant*** 0 ***coup futur***. Cette fonction d'évaluation est généralement choisie de façon heuristique (comme la fonction $\mathcal{H}$ dans l'algorithme A*), en suivant les principes suivants (déjà évoqués en partie dans le cas du morpion 3×3) :

- Plus $\texttt{éval}_0(\texttt{e})$ est grand, plus la situation est favorable pour le joueur-machine ;
- Si $\texttt{e}^-$ est obtenu en inversant l'état e (les pièces et le joueur qui doit joueur sur l'état[6]) alors $\texttt{éval}_0(\texttt{e}^-) = -\texttt{éval}_0(\texttt{e})$ (i.e., $-\texttt{éval}_0$ est la fonction d'évaluation du joueur humain s'il joue de la même façon que la machine) ;
- $\texttt{éval}_0(\texttt{e}) = 0$ correspond à une situation neutre (ou, du moins, une situation dans laquelle on ignore à qui elle est favorable) ;
- Si $\texttt{e}_f$ est un état final dans lequel la machine a gagné, alors $\texttt{éval}_0(\texttt{e}_f) > 0$ (on choisira $\texttt{éval}_0(\texttt{e}_f) = +\infty$ de façon à privilégier une situation de victoire certaine à une situation simplement favorable) ;
- Si $\texttt{e}_f$ est un état final dans lequel la machine a perdu, alors $\texttt{éval}_0(\texttt{e}) < 0$ (on choisira $\texttt{éval}_0(\texttt{e}_f) = -\infty$) ;
- Si $\texttt{e}_f$ est un état final d'égalité, alors $\texttt{éval}_0(\texttt{e}_f) = 0$.

Une fonction d'évaluation $\texttt{éval}_0$ possible consiste à faire la différence entre le nombre de pièces du joueur-machine et le nombre de pièces du joueur humain. On peut améliorer cela en tenant compte du type de pièce (par exemple, une dame sera beaucoup plus pondérée qu'un pion dans un jeu d'échecs) ou de leurs positions (à Othello, une pièce dans un coin est plus avantageuse qu'une pièce en dehors d'un coin).

6. Ce qui suppose que l'identité de ce joueur soit codé dans l'état ou déductible de l'état.

Exercice 11 *Choisissez un jeu à deux joueurs à somme nulle et information complète. Proposez et discutez une fonction d'évaluation de ce jeu.*

Voici une suggestion de tels jeux : Abalone, l'awalé, les dames, le jeu d'échecs, le jeu de go, le jeu de hex, le jeu du moulin, Othello, puissance 4.

L'algorithme minimax a pour but de choisir un coup à jouer à partir de l'état $\mathtt{e}$ pour maximiser $\mathtt{éval}_c(\mathtt{e})$ — c étant un paramètre entier naturel représentant le « nombre de coups futurs à explorer » — où la fonction $\mathtt{éval}_c$ est définie comme suit :

pour un état final $\mathtt{e}_f$:

$$\mathtt{éval}_c(\mathtt{e}_f) = \begin{cases} -\infty & \text{si } \mathtt{e}_f \text{ est un état perdant pour le joueur-machine} \\ 0 & \text{si } \mathtt{e}_f \text{ est un état d'égalité} \\ +\infty & \text{si } \mathtt{e}_f \text{ est un état gagnant pour le joueur-machine} \end{cases}$$

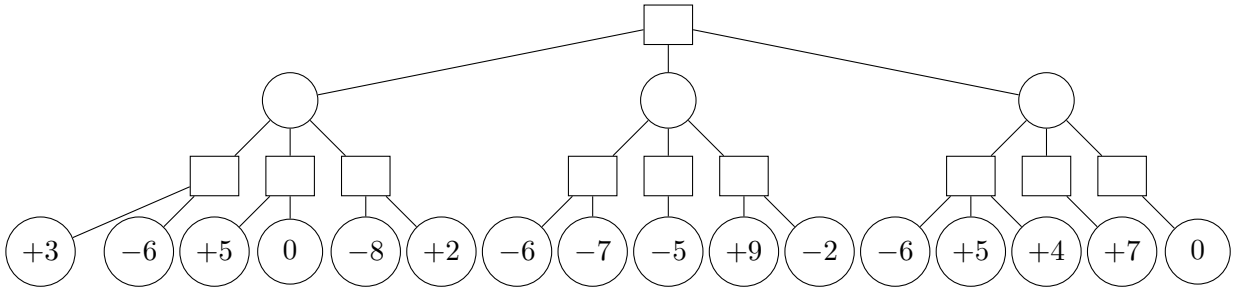
pour un état non final $\mathtt{e}$:

$$\mathtt{éval}_c(\mathtt{e}) = \begin{cases} \mathtt{éval}_0(\mathtt{e}) & \text{si } c = 0 \text{ (utilisation de la fonction } \mathtt{éval}_0) \\ \max\{\mathtt{éval}_{c-1}(s) \mid s \in \mathtt{successeurs}(\mathtt{e})\} & \text{si } c \geqslant 1 \text{ et le joueur courant est le joueur-machine} \\ \min\{\mathtt{éval}_{c-1}(s) \mid s \in \mathtt{successeurs}(\mathtt{e})\} & \text{si } c \geqslant 1 \text{ et le joueur courant est le joueur humain} \end{cases}$$

Cette définition est à mettre en regard des équations (1) et (2).

On peut montrer que plus r est grand, meilleure sera l'évaluation. En revanche, comme le temps de calcul augmente exponentiellement avec p (comme on le verra plus loin), il faut trouver un compromis : un p assez grand pour donner de meilleurs résultats et pas trop pour que le joueur humain ne s'impatiente pas !

Exercice 12 *On considère l'arbre suivant, où les nœuds carrés (resp., ronds) correspondent à des états dans lequel le joueur-machine (resp., le joueur humain) doit jouer et où les valeurs indiquées sont calculées par la fonction* $\mathtt{éval}_0$ *:*



Parcourez l'arbre en profondeur pour remplir chaque case avec la valeur de la fonction $\mathtt{éval}_c$ *où* $c = p - r$ *(*p *est la profondeur de l'arbre,* r *est le rang du nœud dans l'arbre).*

Quel choix fera le joueur-machine dans ce cas ?

Pourquoi parle-t-on d'algorithme minimax ?

La figure 7 récapitule le calcul présenté ci-dessus dans un algorithme. La fonction principale, `minimax` choisit simplement l'état successeur qui maximise la fonction d'évaluation. La fonction d'évaluation calcule $\mathtt{éval}_c$: $\mathtt{évaluation}(c, \mathtt{e}) = \mathtt{éval}_c(\mathtt{e})$.

Si on suppose que la complexité du calcul de $\mathtt{éval}_0(\mathtt{e})$ et celle de la génération des successeurs d'un état sont en $O(1)$, alors la ***complexité*** en temps de l'algorithme minimax est en $O(b^c)$ où b est le facteur de branchement moyen (la moyenne du nombre de coups possibles dans un état) et c est le nombre de coups futurs explorés.

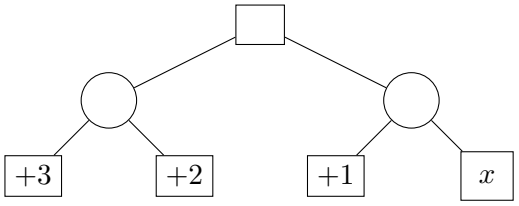
```
Fonction minimax (e : état, c : entier naturel) : état
Variables
    S : ensemble d'états
    s : état
    score, score_max : réel
Début
    S ← successeurs(e) /* ensemble des états possibles suite au coup du joueur-machine */
    score_max ← −∞
    e_sortie ← pas_de_coup_possible /* état indiquant qu'aucun coup n'est possible */
    Pour s ∈ S Faire
        score ← évaluation(c, s)
        Si score ⩾ score_max Alors
            e_sortie ← s
            score_max ← score
        Finsi
    Finpour
    retourner e_sortie
Fin

Fonction évaluation (c : entier naturel, e : état) : réel
Variables
    S : ensemble d'états
    s : état
    score, score_max, score_min : réel
Début
    Si e est un état final (de fin de partie) Alors
        retourner −∞, +∞ ou 0 selon que la partie soit perdue par la machine, gagnée par elle ou nulle
    Finsi
    Si c = 0 Alors
        retourner éval_0(e)
    Finsi
    S ← successeurs(e)
    Si dans l'état e, c'est au joueur-machine de jouer Alors
        score_max ← −∞
        Pour s ∈ S Faire
            score_max ← max(score_max, évaluation(c − 1, s))
        Finpour
        retourner score_max
    Sinon
        score_min ← +∞
        Pour s ∈ S Faire
            score_min ← min(score_min, évaluation(c − 1, s))
        Finpour
        retourner score_min
    Finsi
Fin
```

FIGURE 7 – L'algorithme minimax. e est l'état dans lequel le joueur-machine doit jouer. c est le nombre de coups à l'avance qui sont étudiés.

La complexité dépend donc fortement du facteur de branchement moyen b. On estime que pour le jeu d'Othello, $b \simeq 15$, pour les échecs, $b \simeq 35$ et pour le go, $b \simeq 200$.

4.3 L'élagage $\alpha\beta$

La technique présentée dans cette section permet de diminuer sensiblement le temps de calcul de l'algorithme minimax. L'idée générale est qu'il est inutile d'explorer certains nœuds (et leurs descendants), parce que quelle que soit la valeur du nœud en question, on est sûr qu'elle n'influencera pas le résultat. Par exemple, considérons l'extrait suivant de l'arbre minimax :



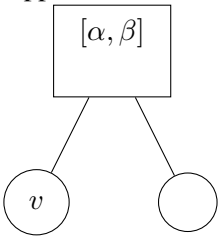
La valeur de la fonction d'évaluation de cet arbre (valeur à mettre à la racine) est donc :

$$\texttt{éval(e)} = \max\{\min\{+3,+2\},\min\{+1,x\}\} = \max\{+2,\min\{+1,x\}\} = +2 \text{ quel que soit } x \in \mathbb{R}$$

Il est donc inutile d'évaluer la valeur de x.

Le principe de l'élagage $\alpha\beta$ est d'associer à chaque nœud un intervalle $[\alpha,\beta]$ de $\overline{\mathbb{R}} = [-\infty,+\infty]$ ($\alpha \leqslant \beta$) tel que la valeur à chercher (`éval(e)`) appartient nécessairement à cet intervalle. Initialement, $[\alpha,\beta] = [-\infty,+\infty]$ mais cet intervalle évolue lors du parcours. En effet :

— Si on est dans un état dans lequel le joueur-machine doit jouer (aussi appelé « état MAX » et symbolisé par un rectangle) et qu'on a évalué la valeur v d'un successeur de cet état (obtenue par appel récursif), trois cas sont possibles selon la « position » de v par rapport à l'intervalle $[\alpha,\beta]$:



1. $v \leqslant \alpha$: rien de spécial dans ce cas, la valeur v n'a pas d'incidence [7] : on passe donc au successeur suivant de l'état MAX courant.
2. $\alpha < v < \beta$: on sait que la valeur du nœud MAX vérifiera $x \in [\alpha,\beta]$ et $x \geqslant \alpha$, ce qui revient à écrire $x \in [v,\beta]$, autrement écrit à remplacer la valeur de α par v ($\alpha \leftarrow v$).
3. $v > \beta$: comme on a un nœud MAX, la valeur calculée dans le nœud sera supérieure ou égale à v et donc, la valeur de ce nœud ne sera pas celle qui sera retenue à la racine de l'arbre. Par conséquent, on peut ne pas considérer les autres successeurs de ce nœud et passer à la suite : on élague les autres successeurs du nœud courant.

— Si on est dans un état dans lequel le joueur humain doit jouer, le raisonnement est dual (on échange α et β et les cas 2 et 3).

L'algorithme 8 détaille tout cela.

Exercice 13 *Reprenez l'arbre de l'exercice 12 et effectuez la recherche minimax avec élagage $\alpha\beta$.*

---

7. Si $v < \alpha$, on sait que v ne sera pas la valeur finale cherchée qui doit être dans l'intervalle, et qu'elle ne modifiera pas l'intervalle : si on sait que $x \in [\alpha,\beta]$, l'information $x \geqslant v$ n'est pas nouvelle. Si $v = \alpha$, l'argument est similaire.

Fonction `minimax_avec_élagage_alpha_bêta` (`e` : `état`, c : `entier naturel`) : `état`
| Obtenue en remplaçant dans l'algorithme `minimax` de la figure 7 :
| « `évaluation`(c, s) » par « `évaluation_alpha_bêta`$(c, s, -\infty, +\infty)$ »

Fonction `évaluation_alpha_bêta` (c : `entier naturel`, `e` : `état`, α : `réel`, β : `réel`) : `réel`
Variables
| `S` : ensemble d'états
| s : `état`
| `score`, `score_max`, `score_min` : `réel`
Début
| **Si** `e` est un état final (de fin de partie) **Alors**
| | **retourner** $-\infty$, $+\infty$ ou 0 selon que la partie soit perdue par la machine, gagnée par elle ou nulle
| **Finsi**
| **Si** $c = 0$ **Alors**
| | **retourner** `éval`$_0$`(e)`
| **Finsi**
| `S` $\leftarrow$ `successeurs(e)`
| **Si** dans l'état `e`, c'est au joueur-machine de jouer **Alors**
| | `score_max` $\leftarrow -\infty$
| | **Pour** $s \in$ `S` **Faire**
| | | `score_max` $\leftarrow \max($`score_max`, `évaluation_alpha_bêta`$(c-1, s, \alpha, \beta))$
| | | **Si** `score_max` $\geqslant \beta$ **Alors** /* *coupure bêta* */
| | | | **retourner** `score_max`
| | | **Finsi**
| | | $\alpha \leftarrow \max(\alpha,$ `score_max`$)$
| | **Finpour**
| | **retourner** `score_max`
| **Sinon**
| | `score_min` $\leftarrow +\infty$
| | **Pour** $s \in$ `S` **Faire**
| | | `score_min` $\leftarrow \min($`score_min`, `évaluation_alpha_bêta`$(c-1, s, \alpha, \beta))$
| | | **Si** `score_min` $\leqslant \alpha$ **Alors** /* *coupure alpha* */
| | | | **retourner** `score_min`
| | | **Finsi**
| | | $\beta \leftarrow \min(\beta,$ `score_min`$)$
| | **Finpour**
| | **retourner** `score_min`
| **Finsi**
Fin

FIGURE 8 – L'algorithme minimax avec coupure $\alpha\beta$.

4.4 Déterminer la fonction d'évaluation

La fonction d'évaluation $\texttt{éval}_0$ doit demander peu de temps de calcul ; il ne s'agit pas de « regarder des coups à l'avance » : c'est l'algorithme minimax qui se charge de cela. Comme pour le choix de la fonction heuristique $\mathcal{H}$ d'A*, il n'y a pas de méthode systématique pour définir une telle fonction [8]. On peut néanmoins s'appuyer sur les idées développées ci-dessous.

Comparaison de deux fonctions $\texttt{éval}_0$. Pour un jeu donné, considérons deux fonctions $\texttt{éval}_0$ et $\texttt{éval}'_0$ pour l'évaluation des feuilles de l'arbre minimax. On peut comparer ces deux fonctions en faisant jouer la machine contre elle-même, i.e., le programme avec la fonction $\texttt{éval}_0$ et celui avec la fonction $\texttt{éval}'_0$. Si $\texttt{éval}_0$ « bat » $\texttt{éval}'_0$ quel que soit le joueur qui commence, alors $\texttt{éval}_0$ sera meilleur que $\texttt{éval}'_0$. On notera alors $\texttt{éval}_0 \succ \texttt{éval}'_0$. À moins qu'il y ait des choix aléatoires dans une des fonctions, il suffit de faire deux parties : une pour laquelle le « joueur $\texttt{éval}_0$ » commence, une pour laquelle le « joueur $\texttt{éval}'_0$ » commence.

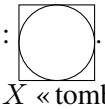
On notera qu'il est possible que ni $\texttt{éval}_0 \succ \texttt{éval}'_0$ ni $\texttt{éval}'_0 \succ \texttt{éval}_0$ (noté $\texttt{éval}_0 \sim \texttt{éval}'_0$) : $\succeq$ n'est pas une relation d'ordre. En revanche, on supposera que $\succ$ est transitive : si $\texttt{éval}_0 \succ \texttt{éval}'_0$ et $\texttt{éval}'_0 \succ \texttt{éval}''_0$ alors $\texttt{éval}_0 \succ \texttt{éval}''_0$ (c'est une hypothèse).

On peut appliquer cela pour chercher une fonction d'évaluation de la façon suivante. On suppose qu'on a défini une fonction d'évaluation $\texttt{éval}_0^\lambda$ dépendant d'un paramètre λ qui peut être constitué d'une table de nombres. Par exemple, pour un jeu d'échecs, λ peut être la fonction qui à une position et à une pièce associe un réel, par exemple $\lambda(\text{c6}, ♘) = 5$. Dans un état $\texttt{e}$, on peut calculer $\texttt{éval}_0^\lambda(\texttt{e})$ par exemple en faisant la somme de ces valeurs :

$$\texttt{éval}_0^\lambda(\texttt{e}) = \sum_{p\text{:position sur laquelle une pièce est présente}} \lambda(p, \texttt{pièce\_sur}(p, \texttt{e}))$$

où $\texttt{pièce\_sur}(p, \texttt{e})$ est le type de pièce à la position p dans l'état $\texttt{e}$. Le problème est alors ramené au problème de la recherche d'un « bon » paramètre λ. Idéalement, on cherche une paramètre optimal λ^*, i.e., celui pour lequel on a, pour tout λ, $\texttt{éval}_0^{\lambda^*} \succ \texttt{éval}_0^\lambda$ ou $\texttt{éval}_0^{\lambda^*} \sim \texttt{éval}_0^\lambda$. Une façon de faire non optimale — notamment à cause de potentiels optimums locaux — consiste à partir d'un λ donné et à faire aléatoirement une « petite modification » pour obtenir un paramètre λ'. On compare ensuite λ et λ'. Si $\texttt{éval}_0^{\lambda'} \succ \texttt{éval}_0^\lambda$, on prend λ' comme nouvelle valeur pour λ ($\lambda \leftarrow \lambda'$) et on recommence. On peut s'arrêter par exemple quand on n'a pas eu d'amélioration 5 fois de suite. Cette description d'un processus pour trouver un bon $\texttt{éval}_0^\lambda$ est évidemment très améliorable.

Application de la méthode de Monte-Carlo Une autre approche pour déterminer $\texttt{éval}_0$ s'appuie sur la méthode de Monte-Carlo, laquelle permet d'estimer une probabilité (ou, plus généralement, une espérance) grâce à des tirages aléatoires.

Un bref rappel sur cette méthode indépendamment de la problématique des jeux peut s'avérer utile. Considérons le problème suivant : dans le plan euclidien, soit un carré et le cercle inscrit dans ce carré :  . Soit une variable aléatoire X sur ce carré de distribution uniforme. On s'intéresse à la probabilité p^* que X « tombe » dans le disque. Une façon de faire consiste à générer (pseudo-)aléatoirement des positions du carré, à compter le nombre de points n_d « tombant » dans le disque et à le diviser par le nombre total n_t de lancers aléatoires. Le rapport $\dfrac{n_d}{n_t}$ est une approximation de la probabilité p^* d'autant meilleure que n_t est grand [9].

Étant donné un état du jeu $\texttt{e}$, on cherche à estimer $p^*(\texttt{e})$, la probabilité pour le joueur de gagner sachant qu'il est dans un état $\texttt{e}$ [10]. Si $p^*(\texttt{e}) > 0{,}5$, les « chances » sont favorables au joueur, si $p^*(\texttt{e}) < 0{,}5$, elles lui sont défavorables et si $p^*(\texttt{e}) = 0{,}5$, la situation est équilibrée. L'idée est alors que $\texttt{éval}_0^*(\texttt{e}) = 2p^*(\texttt{e}) - 1$ est une fonction d'évaluation intéressante : celle qui mène à un choix maximisant les chances de gagner (sous l'hypothèse

8. Du moins, pas de méthode générale connue de l'auteur de ces notes de cours.

9. Plus précisément, il s'agit d'une convergence en probabilité : pour tout $\varepsilon > 0$, il existe un entier $N \geqslant 0$ tel que pour tout $n_t \geqslant N$, la probabilité que $|p - \frac{n_d}{n_t}| \leqslant \varepsilon$ est nulle. Cela peut aussi être vu comme un « moyen expérimental » pour estimer π, puisque $p^* = \frac{\pi}{4}$.

10. Pour bien définir mathématiquement cette probabilité, il faudrait faire des hypothèses que nous négligerons de faire ici : cet exposé se veut plus intuitif que précis.

— très — simplificatrice d'un choix aléatoire de la part de l'adversaire...). L'idée est alors d'estimer $p^*(\mathtt{e})$ par la méthode de Monte-Carlo. Pour cela, à partir de l'état $\mathtt{e}$, on génère aléatoirement des parties jusqu'à leurs termes et on compte le rapport $p(\mathtt{e})$ entre le nombre de parties gagnées et le nombre de parties jouées. $p(\mathtt{e})$ constitue alors une estimation de $p^*(\mathtt{e})$ et on prendra $\mathtt{éval}_0(\mathtt{e}) = 2p(\mathtt{e}) - 1$.

Cette approche a permis d'améliorer sensiblement la performance de ce type de jeux.

*****

Fin de la première partie