Intelligence Artificielle

Méthodes de Résolution de Problèmes

Patrick Esquirol
INSA Toulouse

version du 29/01/2019

I Introduction

Domaine "Résolution de Problèmes" en I.A. :

- Modèles et algorithmes permettant d'exploiter des connaissances heuristiques sur un problème. Méthodes de recherche arborescente

  Application : problèmes combinatoires, robotique, productique, transport, démonstration automatique, jeux à 2 joueurs …

- Problèmes de décision, problèmes d'optimisation

  problèmes sont bien définis (bien spécifiés). **pas de hasard**.

- Complexité algorithmique

  problèmes difficiles : il n'existe **pas d'algorithme de résolution à la fois général et efficace**.

Exemples :

• jeux à 2 joueurs (ex : échecs, go, Othello…) (IA)

2 joueurs jouent à tour de rôle. Il faut parvenir à un état terminal « gagnant » pour l'un des 2 joueurs.

• démonstration automatique, satisfiabilité de formules booléennes (problème SAT) : (IA)

montrer qu'il existe une interprétation qui rend simultanément vraies un ensemble de formules.

• raisonnement temporel, planification d'actions (IA)

Il faut trouver une séquence d'actions (ex : une trajectoire dans l'espace des configurations d'un robot) qui respecte les contraintes sur l'état (énergie restante, configuration du robot) et sur les actions (déplacements, temps et ressources limités).

• affectation de ressources, ordonnancement (RO, IA)

Il faut trouver quelles ressources affecter et/ou quand commencer chaque tâche d'un ensemble

• transports, tournées de véhicules, pb du voyageur de commerce (RO)

Il faut trouver un(des) circuit(s) passant par toutes les villes d'une carte en minimisant la distance totale parcourue, l'énergie dépensée, le temps de parcours …

Algorithmes d'IA « classique » :

- recherche d'un chemin optimal dans un graphe d'états — A*
- recherche d'un sous-graphe optimal dans un graphe de sous-problèmes — AO*
- recherche d'un coup optimal dans un arbre de jeu à 2 joueurs — Minmax, $\alpha\beta$

Dans chaque cas, la résolution fait appel à **un algorithme de recherche arborescente**

- à chaque étape un choix est effectué parmi plusieurs
- le choix peut être guidé par une heuristique, mais sans garantie d'optimalité
- il n'y a pas d'ORACLE capable de prédire le meilleur choix à l'avance : échec $\Rightarrow$ retour arrière
- le temps de résolution dépend de la taille de l'espace de recherche

heuristique = méthode approchée fournissant **d'assez bonnes solutions** par rapport à une méthode exacte trop coûteuse à mettre en œuvre.

II Recherche heuristique dans les graphes d'états

II.1 Formulation générale

Soit :

- U l'espace des états d'un système (ex : un robot, un jeu, un système de production …). U est discret fini ou infini. On recherche un chemin d'un état initial à un état *terminal*, satisfaisant une condition donnée
- u_0 l'état initial
- T l'ensemble des états terminaux (**ou états buts**)

En chaque état, des règles (ou des actions) sont applicables ; chaque règle réalise un changement d'état, si possible pour se rapprocher d'un état but :

r (règle applicable, ou action)

u : état courant → **v** : état suivant

notation : **v = suc(r, u)** (notation fonctionnelle) **transition(r, u, v)** (notation logique)

k(u,v) = coût de la règle (transition) de u à v k est constant ou bien dépend uniquement de u et v.

Problème = **trouver une séquence** $(r_1\ r_2\ \dots\ r_{n-1}\ r_n)$ permettant de relier u_0 à un des états buts de T ou

⇔ **trouver une séquence d'états admissibles** $(u_0\ u_1\ \dots\ u_{n-1}\ u_n)$ tel que la **somme des coûts** $\sum_{i=1}^{i=n} k(u_{i-1}, u_i)$ soit **minimale**.

Des **contraintes** pèsent sur les états et/ou les actions :

états **admissible** (vs états interdits)

règles/actions **légales**.

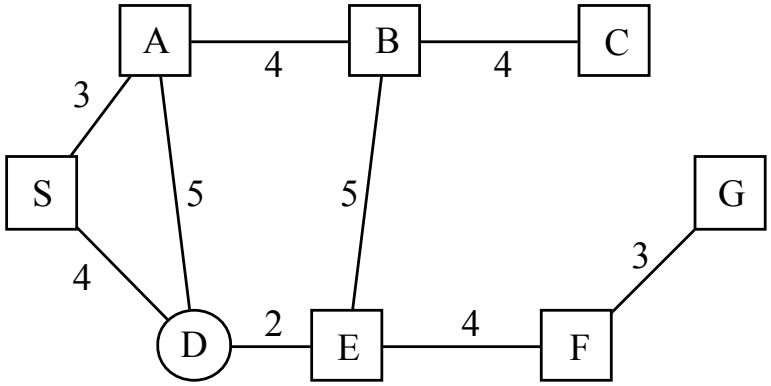
On note :

- S(u) les **successeurs** admissibles d'un état u : {v / ∃r v = suc(r,u)} ; par extension, S(U) désigne l'union des successeurs de tous les états d'un ensemble U.
- D(u) les **descendants** de u : S(u) ∪ S(S(u)) ∪ ...
- S_{-1} (u) les **prédécesseurs** de u {v / ∃r u = suc(r,v)} ; par extension, S_{-1}(U) désigne les prédécesseurs de tous les états d'un ensemble U.
- D_{-1} (u) les **ascendants** de u : S_{-1}(u) ∪ S_{-1} (S_{-1} (u)) ∪ ...

Représentation du graphe d'états

Exemple de problème :

rechercher le chemin le plus court entre S et G dans le réseau ci-contre :



Un réseau routier

Etat = séquence des villes déjà traversées

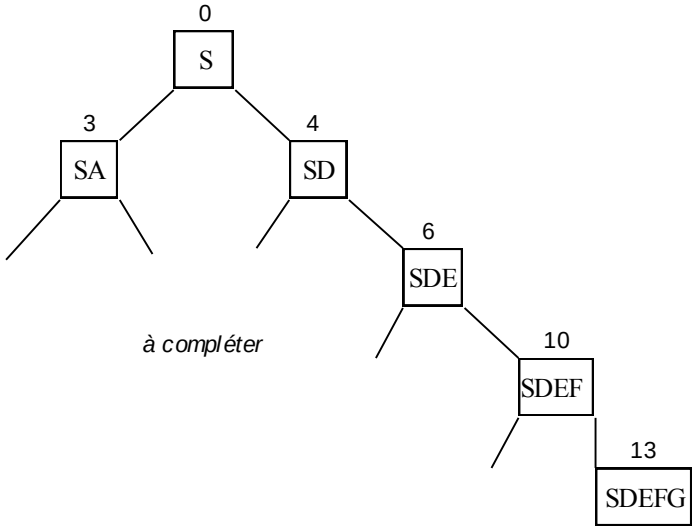
- état initial : une seule ville traversée (la ville de départ) : {S}
- état intermédiaire : une séquence (chemin partiel) contenant les **villes déjà traversées**, ex : {S,D}
- état terminal : une séquence qui se termine par la ville d'arrivée G, ex : {S,D,E,F,**G**}, {S,A,B,E,F,**G**}
- Espace d'états : **ensemble de tous les chemins partiels issus de S**.

Coût entre 2 états u et v = *longueur [temps, consommation]* pour aller de u à v.

Convention : pas de coût négatif ; si pas d'arc entre 2 villes u et v $\Leftrightarrow$ k(u,v) = ∞.

Calcul du coût cumulé :

Etat	Coût
S	0
SD	4
SDE	6
SDEF	10
SDEFG	13



Espace d'états liés à la recherche d'un chemin entre S et G

Différences entre les algos de plus court chemin (Théorie des Graphes) et les algorithmes vus en I.A.?

- en Théorie des Graphes, le graphe sur lequel on cherche un chemin le plus court peut être grand (ex : Carte Europe) mais il est connu à l'avance. Les algorithmes calculent le plus court chemin (Bellman ou Dijkstra) et peuvent être **matriciels**.

- en I.A., le **graphe des états d'un système** n'est pas donné à l'avance, on se donne seulement les règles de transition entre états ; l'espace est partiellement **généré au fur et à mesure** de l'exploration à l'aide d'algos utilisant des **listes** et/ou des **files de priorité**.

II.2 Algorithmes sans prise en compte des coûts

Il n'y a pas de critère d'optimisation et pas de coûts sur les actions. On cherche à atteindre un état terminal.

II.2.1 Recherche aveugle (dans un graphe sans circuits)

On code chaque état par une situation (sans mémoriser le chemin suivi).

Exemple : **état** d'un robot porteur dans un espace discrétisé 3D, on aura : E = {X,Y,Z,O, E, Items}avec :

X,Y,Z = variables de positions 3D, 0 = orientation (nord, sud, est, ouest), E= niveau d'énergie(1..10),

Items = liste d'objets portés par le robot.

On désire simplement parvenir à un état terminal (exploration).

a/ *parcours en profondeur d'abord*

1 initialiser une pile d'états P vide

2 empiler(u_0, P)

2 tantque P $\neq \varnothing$ et sommet(P) $\notin$ T

$u_i \leftarrow$ sommet(P)

dépiler(P)

déterminer S(u_i)

pour chaque s $\in$S(u_i)

empiler(s, P)

finpour

fintantque

3 si P $\neq \varnothing$ alors sommet(P) est un état terminal

sinon pas de solution

finsi

b/ *parcours en largeur d'abord*

Même algo qu'en a/ en utilisant **une file** au lieu d'une pile (**remplacer "empiler"** par **"enfiler"**)

c/ *parcours en profondeur limitée*

Compromis entre a/ et b/ ; un niveau est associé à chaque état (nombre d'actions réalisées depuis u_0).

Le niveau de u_0 est 0.

On utilise une liste P ordonnée par niveau décroissant (la tête de liste est l'état de plus grand niveau).

Soit $niv_0 = \min_{u \in P} niv(u)$ = niveau de l'état situé en fin liste

On choisit un état u^* de P, de plus grand niveau, mais tel que : $niv(u^*) - niv_0 \leq d$

d est la limite de profondeur. Si d est très grand, on retrouve l'algorithme a/, si d=0 on retrouve l'algorithme b/

Bilan de a/, b/, c/ :

Ces algorithmes sont simples mais bouclent si le graphe d'état comporte des circuits (suite d'actions/de règles qui font revenir dans un état déjà exploré).

II.2.2 Algorithmes de recherche sans bouclage

L'hypothèse de non-existence de circuits n'est pas réaliste. **Il faut mémoriser les états déjà atteints** pour éviter de les redévelopper indéfiniment.

Première option : on associe à chaque état la séquence des états antérieurs (chemin) depuis l'état initial.

On manipule donc des états qui sont des **chemins partiels**.

Notation :

P = pile de chemins partiels non encore développés. On développe systématiquement le premier chemin partiel situé en haut de la pile P : sommet(P).

sommet(P) est un chemin partiel de la forme C={u_0, u_1, …, u_i} ;

u_i= dernier(C) = dernier(sommet(P)) est l'état courant à développer

a'/ ***parcours en profondeur d'abord***

1 Initialiser une pile P contenant le seul chemin partiel $C_0 = \{u_0\}$

2 tantque $P \neq \varnothing$ et dernier(sommet(P)) $\notin T$

C ← sommet(P) u_i ← dernier (C)

dépiler(P)

déterminer $S(u_i)$

pour chaque $s \in S(u_i)$

si $s \notin \{u_0, u_1, \ldots, u_{i-1}\}$ /* *il faut éviter de boucler sur un état déjà atteint*/*

alors C' ← $\{u_0, u_1, \ldots, u_i, s\}$

empiler(C', P)

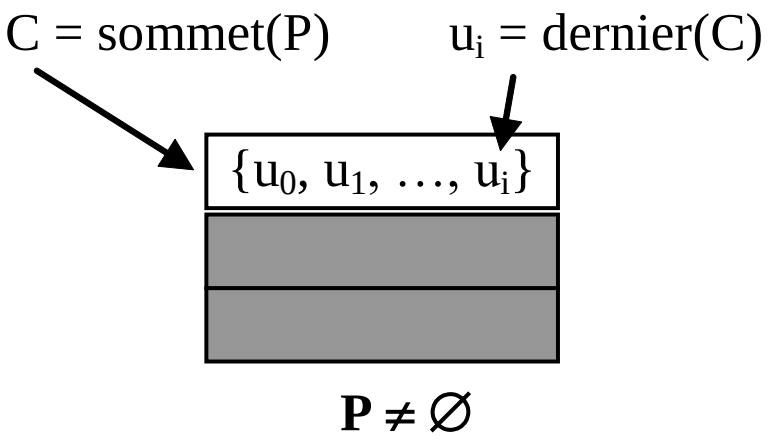
finsi

finpour

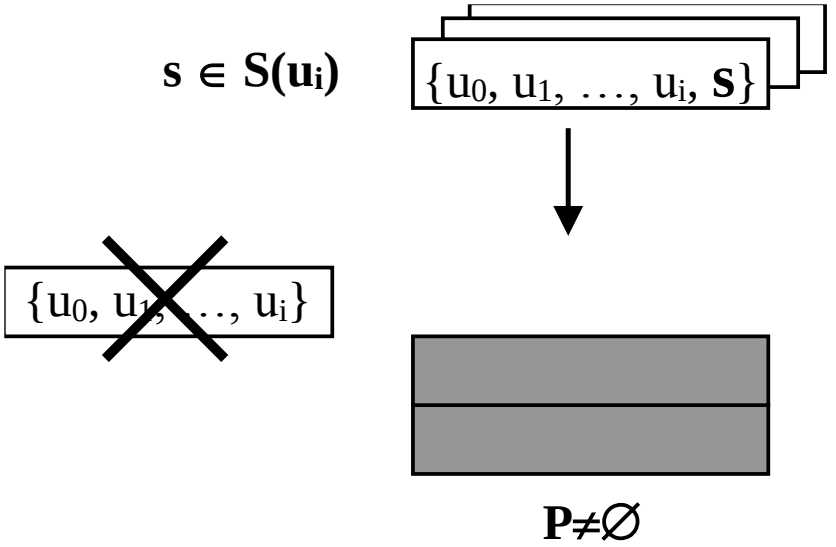
fintantque

3 si $P \neq \varnothing$ alors sommet(P) est un chemin solution sinon pas de solution

finsi



une pile de chemins partiels



traitement de la pile

b'/ ***parcours en largeur d'abord***

Même algo qu'en a'/ en remplaçant la pile par une file

c'/ ***parcours en profondeur limitée***

Compromis entre a' et b' : on évite de s'enfoncer dans la génération de chemins partiels de plus en plus longs.

Même principe que c/ avec un pile P de chemins partiels.

Critique des algorithmes a'/, b'/, c'/ :

1/ P peut contenir plusieurs chemins partiels de même extrémité => on développe autant de fois un chemin partiel arrivant en u_i qu'il y a de chemins de u_0 à u_i … Il faudrait ne garder que « le meilleur » …

2/ La représentation des éléments stockés dans la pile (chemins partiels) est trop coûteuse.

Gérer des piles de chemins partiels de taille croissante peut avoir un coût prohibitif.

Idée : on ne conserve qu'un seul exemplaire de chaque chemin partiel aboutissant à u_i, celui qui est **le plus court** (en termes de nombre d'actions effectuées, ou de coût total cumulé de ces actions, ou un chemin partiel quelconque en cas d'égalité).

Pour cela il suffit de mémoriser pour chaque état atteint u_i son « **meilleur prédécesseur** », c-à-d l'état **pere(u_i)** qui le précède sur le chemin **le plus court** connu depuis l'état initial.

C'est le principe des deux algorithmes qui suivent (A et A*).

II.3 Algorithmes avec prise en compte des coûts

II.3.1 Recherche de type *"meilleur d'abord"*

L'ALGORITHME A (DIJKSTRA)

Par hypothèse :

- **S(u)** fini (le nombre de règles applicables en chaque état est fini). **Développer u**, c'est déterminer S(u).
- **$k(u_i, u_r)$**, le coût d'un chemin $\{u_i,\ldots,u_r\}$ est fini : somme des coûts associés aux actions (u_i,u_{i+1}) du chemin.
- **k*(u,v)** = coût du meilleur chemin (minimal) reliant u à v ; la valeur est infinie si aucun chemin de u à v.

On définit :

- **$g^*(u_i)$** = $k^*(u_0 ,u_i)$ = coût du chemin optimal allant de u_0 à u (au départ il est inconnu)
- **$g(u_i)$** coût du **meilleur chemin connu** de u_0 à u_i. On aura $g(u_i) = g^*(u_i)$ lorsqu'à une itération donnée, le meilleur chemin entre u_0 et u_i aura été trouvé.
- **pere(u_i)** = pointeur indique l'état qui précède u_i sur le meilleur chemin connu depuis u_0.

ALGORITHME **A** : "*MEILLEUR D'ABORD*" (⇔ Dijkstra pour graphes infinis)

1 $P \leftarrow \{u_0\}$; $Q \leftarrow \varnothing$; $g(u_0) \leftarrow 0$; /* initialisations */

2 tantque $P \neq \varnothing$ et sommet(P) $\notin T$

  $u_i \leftarrow$ sommet(P) ; dépiler(P)

  $Q \leftarrow Q \cup \{u_i\}$

  déterminer $S(u_i)$

  pour chaque $s \in S(u_i)$

    si $s \notin P \cup Q$ ou si $g(s) > g(u_i) + k(u_i,s)$

    alors $g(s) \leftarrow g(u_i) + k(u_i, s)$ /* actualisation de g(s) */

      pere(s) $\leftarrow u_i$

      insérer s dans P par ordre ↗ de g /* sinon s n'est pas mémorisé */

    finsi

  finpour

fintantque

3 si $P \neq \varnothing$, alors u_i = premier(P) et le chemin $\{u_0, \ldots, pere(pere(u_i)), pere(u_i), u_i)$ est un chemin optimal.

    sinon il n'y a pas de solution

finsi.

L'algorithme A classe les états de P *pendants* (non encore développés) par **ordre croissant de $g(u_i)$**. Il développe en priorité l'état u_i de valeur $g(u_i)$ minimale. Q mémorise les états déjà développés.

A tout instant P contient un sous-chemin d'un chemin optimal. Tôt ou tard ce sous-chemin sera développé.

Démonstration informelle :

Lorsque A termine (en 3) avec P non vide, le sommet de P est au bout du chemin le plus court depuis u_0 (les autres états de P ne peuvent fournir qu'un coût supérieur).

Application sur l'exemple de la page 7

Initialisation

P	Q	Meilleurs coûts	Peres
S	∅	g(S)=0	

Développement de S : successeurs de S = {A,D}

P	Q		Peres
A, D	{S}	g(S)=0 ; **g(A)=3** ; g(D)=4	pere(A)=S ; pere(D)=S

Développement de A : successeurs de A = {B,D}

Le chemin SAD n'est pas intéressant car on a déjà pu arriver en D à moindre coût : g(A) + k(A,D) > g(D) (3+5 > 4)

P	Q	Meilleurs coûts	Peres
D, B	{S, A}	g(S)=0 ; g(A)=3 ; g(D)=4 ; g(B)=7	pere(A)=S ; pere(D)=S ; pere(B)=A

Développement de D : successeurs de D = {A, E} A est écarté car g(D) + k(D,A) > g(A) (4+5 >3)

P	Q	Meilleurs coûts	Peres
E, B	{S, A, D}	g(S)=0 ; g(A)=3 ; g(D)=4 ; g(B)=7 ; **g(E)=6**	pere(A)=S ; pere(D)=S ; pere(B)=A ; pere(E)=D

Développement de E : successeurs de E = {B, F} B est écarté car g(E) + k(E,B) > g(B) (6+5 >7)

P	Q	Meilleurs coûts	Peres
B, F	{S,A,D,E}	g(S)=0 ; g(A)=3 ; g(D)=4 ; **g(B)=7** ; g(E)=6 ; g(F)=10	pere(A)=S ; pere(D)=S ; pere(B)=A ; pere(E)=D ; pere(F)=E

Développement de B : successeurs de B = {C, E} E est écarté car g(B) + k(B,E) > g(E) (7+5 >6)

P	Q	Meilleurs coûts	Peres
F, C	{S,A,D,E,B}	g(S)=0 ; g(A)=3 ; g(D)=4 ; g(B)=7 ; g(E)=6 ; **g(F)=10** ; g(C)=11	pere(A)=S ; pere(D)=S ; pere(B)=A ; pere(E)=D ; pere(F)=E ; pere(C)=B

Développement de F : successeurs de F = {G}

P	Q	Meilleurs coûts	Peres
C, G	{S,A,D,E,B,F}	g(S)=0 ; g(A)=3 ; g(D)=4 ; g(B)=7 ; g(E)=6 ; g(F)=10 ; **g(C)=11** ; g(G)=13	pere(A)=S ; pere(D)=S; pere(B)=A ; pere(E)=D ; pere(F)=E ; pere(G)=F

Développement de C : successeurs de C = {}

P	Q	Meilleurs coûts	Peres
G	{S,A,D,E,B,F, C}	g(S)=0 ; g(A)=3 ; g(D)=4 ; g(B)=7 ; g(E)=6 ; g(F)=10 ; g(C)=11 ; **g(G)=13**	pere(A)=S ; pere(D)=S; pere(B)=A ; pere(E)=D; pere(F)=E ; pere(G)=F

STOP, le but, G, est atteint ; le meilleur chemin est SDEFG, de coût 13.

La procédure A est la plus évoluée lorsqu'on ne dispose d'aucune autre information sur la fonction de coût.

Dans certains problèmes, des connaissances spécifiques permettent **d'estimer le coût du chemin restant** à parcourir jusqu'au but. C'est l'idée qu'utilise l'algorithme A*.

L'ALGORITHME A*

Principes et notation

- h*(u) = coût du plus court chemin entre u et un état terminal = ***information parfaite***
- **h(u) = estimation heuristique de h*(u)**. h*(u) – h(u) mesure la qualité de h (calculable a posteriori).
- f*(u) = g*(u) + h*(u) : longueur du chemin optimal (le plus court) partant de l'état initial u_0 , passant par u et allant à un état terminal $t \in T$.
- f(u) = g(u) + h(u) = estimation heuristique de f*(u).

A* partitionne les états engendrés en 2 sous-ensembles :

- P = ensemble des états *pendants* (non encore développés),
- Q = ensemble des états *clos* (déjà développés).

P est **totalement ordonné** par f ↗

Les états ayant même valeur de f, sont ordonnés par g↘ (ou ce qui revient au même par h↗).

En cas d'égalité de f et de g (et donc de h), un état terminal est placé devant un état non terminal ; sinon l'ordre est arbitraire.

Pour tout état u de P ou de Q, **A* ne retient qu'un seul chemin de u_0 à u**, en associant à u un pointeur père(u) sur l'état qui le précède sur ce chemin.

On note : chemin(u) = {u_0 , ... , père(père(u)), père(u), u}

1 $P \leftarrow \{u_0\}$; $Q \leftarrow \varnothing$; $g(u_0) \leftarrow 0$; par défaut : $g(u_i) \leftarrow \infty \ \forall i \neq 0$ /* Initialisations */

2 tantque $P \neq \varnothing$ et sommet(P) $\notin$ T

u $\leftarrow$ premier(P) ; dépiler(P)

$Q \leftarrow Q \cup \{u\}$

déterminer S(u)

pour chaque s $\in$ S(u)

si s $\notin P \cup Q$ /*s est nouveau (jamais atteint par l'algorithme)*/

ou si g(s) > g(u) + k(u, s) /*pas nouveau (il est déjà dans P ou Q) mais atteint par un meilleur chemin*/

alors g(s) $\leftarrow$ g(u) + k(u,s) /*alors on doit mémoriser le coût du chemin arrivant en s via u */

f(s) $\leftarrow$ g(s) + h(s)

pere(s) $\leftarrow$ u /* et mémoriser que le meilleur prédécesseur pour arriver en s c'est u */

inserer s dans P d'abord selon f ↗ et en cas d'égalité selon g↘

finsi /* sinon s n'est pas mémorisé */

finpour

fintantque

3 si $P \neq \varnothing$, alors t = premier(P) $\in$ T et $\{u_0, \ldots, pere(pere(u_i)), pere(u_i), u_i\}$ est un chemin optimal.

sinon il n'y a pas de solution

finsi

II.3.2 Propriétés de A*

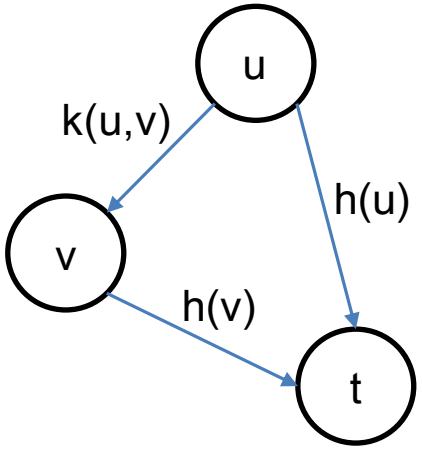
Propriétés de h(u)

- h est une heuristique ***presque parfaite*** ssi : $\forall u, \forall v, \quad h(u) < h(v) \Rightarrow h^*(u) < h^*(v)$

  (h ordonne les états de la même façon que h*)

- h est ***monotone*** ssi : $\forall u, \forall v \in S(u), \quad h(u) - h(v) \leq k(u,v)$

  (« inégalité triangulaire »)



- h est ***coïncidente*** ssi : $\forall t \in T \quad h(t) = 0$
- h est ***minorante*** ssi : $\forall u, \quad h(u) \leq h^*(u)$

Tout heuristique monotone et coïncidente est également minorante.

Preuve : soit $\{u, v_1, v_2, .., v_q, t\}$ un chemin optimal de u vers un état terminal.

Comme h est monotone on peut écrire :

$h(u) - h(v_1) \leq k(u, v_1)$

$h(v_1) - h(v_2) \leq k(v_1, v_2)$

$... \leq ...$

$h(v_q) - h(t) \leq k(v_q, t)$

$h(u) - h(t) \leq k(u, v_1) + k(v_1, v_2) + k(v_2, v_3) + \ldots + k(v_q, t) = h^*(u)$ puisque le chemin de u à t est optimal.

Comme h est coïncidente, $h(t) = 0$, et donc $\mathbf{h(u) \leq h^*(u)}$ CQFD

Terminaison

Pour tout **graphe fini** et pour toute heuristique h définie positive ou nulle, **A* termine si k est définie positive ou nulle, soit sur une solution (optimale) soit parce que P vide**.

Preuve : le nombre d'états est fini ⇒ le nombre de chemins sur le graphe d'états est fini…
Un état déjà développé (dans Q) ne peut revenir en P que si l'on trouve un nouveau chemin qui y mène ayant un coût strictement inférieur à celui déjà connu. Eventuellement tous les chemins menant à cet état seront trouvés (si par malchance on les trouve dans un ordre améliorant à chaque fois le coût). Lorsque le meilleur est trouvé, cet état reste définitivement dans Q et ne réapparait plus dans P. Le nombre de chemins possibles de u_0 à cet état est fini, le nombre d'itérations l'est donc aussi. Nécessairement, s'il existe, un état terminal apparaîtra en tête de P au bout d'un nombre fini d'itérations ; sinon l'arrêt se produira avec P vide (pas de solution) après avoir exploré tous les chemins du graphe d'états.

Dans le cas d'un graphe infini, il est nécessaire d'établir d'abord la propriété suivante : **s'il existe une solution optimale alors, à chaque itération de A*, il existe toujours dans P un état situé à l'extrémité d'un chemin partiel optimal**.

Proposition

Dans le cas d'un graphe fini ou infini contenant au moins une solution (chemin de u_0 à t $\in$ T), à chacune des itérations de A*, à tout chemin optimal de u_0 à t correspond au moins un état pendant u_i tel que **$g(u_i) = g^*(u_i)$.**

Preuve: soit C = (u_0, u_1, u_2, ... t) un chemin optimal. L'état u_0 est d'abord développé (passage dans Q). Donc u_1 sera trouvé et sera soit (1) dans P soit (2) dans Q, mais alors u_2 sera aussi dans P ou dans Q, ... etc. t ne pourra pas être dans Q (puisque A* s'arrête lorsqu'il le rencontre en tête de P). Soit u_i le premier état de C à passer dans P à une certaine itération ; les états u_0, u_1, u_2, ... u_i ayant été déjà développés, ce chemin est connu de A*, et il est optimal : $g(u_i) = g^*(u_i)$

Dans le cas d'un graphe infini contenant au moins une solution, si la fonction h est bornée sur cette solution, alors A* s'arrête et fournit la solution.

☞ **par contre, en l'absence de solution, et dans le cas d'un graphe infini, A* peut ne jamais s'arrêter …**

Admissibilité

Si h est minorante, à la fin de chaque itération de A*, l'état û situé au sommet de P est tel que : $f(\hat{u}) \leq f^*(u_0)$.

Preuve

Par définition û est défini par $f(\hat{u}) = \min\{f(u) \mid u \in P\}$.

D'autre part à tout moment, s'il existe une solution finie, il existe toujours dans P un état u_i sur un chemin optimal tels que $g(u_i) = g^*(u_i)$. û étant en tête de P on a $f(\hat{u}) \leq f(u_i)$ (par définition du classement des états de P par ordre croissant)

h étant minorante, on a : $h(u_i) \leq h^*(u_i)$

$f(u_i) = g(u_i) + h(u_i) = g^*(u_i) + h(u_i) \leq g^*(u_i) + h^*(u_i) = f^*(u_i) = f^*(u_0)$

et $f(\hat{u}) \leq f(u_i)$

donc $f(\hat{u}) \leq f^*(u_0)$ (CQFD)

Comme A* s'arrête, il finit nécessairement par rencontrer un état û terminal et chemin(û) est optimal, donc A* est admissible.

Remarque : pour une fonction h minorante, le sous-graphe exploré se limite aux seuls états $\{u \mid g^*(u) + h(u) \leq f^*(u_0)\}$

A* est également admissible si h est monotone et si h coincidente.

Complexité

Dans le pire des cas : $O(2^N)$, chaque état étant développé autant de fois qu'il y a de chemins possibles entre et u_0 et cet état.

Si h est minorante, la complexité peut descendre à $O(N^2)$.
Le choix de l'état à développer est û tel que f(û) $\geq$ max(f(v)) v $\in$ Q ;
sinon parmi tous les états u tels que f(u) < max(f(v)), il faut développer celui à g(u) minimal.

Si h est monotone, tout état u développé est tel que g(u) = g*(u). Donc u est développé au plus une fois. Donc complexité = $O(N)$. Attention N peut être très grand :
exemples : N = 9! pour le jeu du taquin N = 4,3 10^{19} pour le Rubik's cube.

Une autre caractérisation de la complexité consiste à utiliser la longueur en moyenne M des chemins optimaux. Dans ce cas, A* est $O(e^M)$

Conclusion : en pratique on évite A* dans les cas où la séquence optimale comporte plus d'une trentaine d'états.

Cependant, cette propriété s'applique à la classe entière des algorithmes admissibles. Tout autre algorithme disposant d'une heuristique moins informée est nécessairement plus mauvais :

A* reste le meilleur algorithme pour cette classe de problèmes.

II.3.3 Mise en œuvre et complexité pratique

A* doit être programmé de manière **générique**. Ce qui particulier à un problème donné, c'est :

- **E** : la structure de données représentant un état, ex : `[[a,b,c,],[d,e,f],[g,h,_]]`
- **cle(u)** : la fonction qui permet d'identifier de manière unique un état, ex : « `abcdefgh_`»
- **S(u)** : la fonction de transition qui détermine les états suivants d'un état
- **k(u, v)** : la fonction de coût pour passer de u à v $\in$ S(u)
- **h(u)** : la fonction heuristique qui estime la distance entre u et un état terminal.

A chaque étape de l'algorithme deux besoins distincts pèsent sur la complexité :

- **accéder au « meilleur » état û** à développer dans P (de valeur $f(\hat{u})$ minimale)

  On peut le faire en $O(1)$, si on gère **P** à l'aide d'une **file de priorité** (un tas binaire) ordonnée selon les valeurs de f puis de h (en cas d'égalité de f)

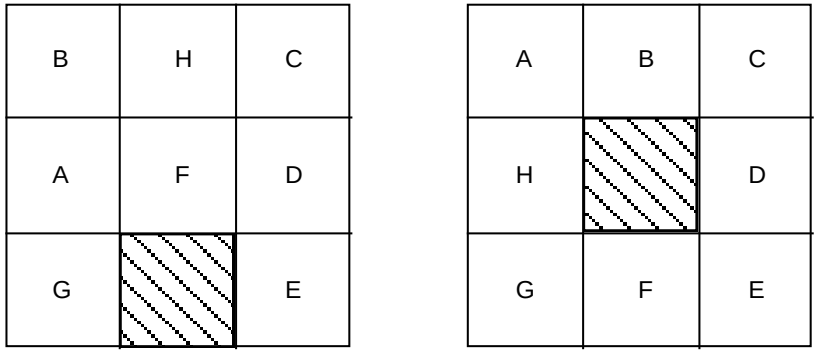
- décider du sort de chaque successeur généré $v \in S(u)$: il faut rechercher v dans P et sinon dans Q, de façon à ne garder qu'un seul exemplaire de v (celui qui minimise f)

Il faut éviter une recherche en table (de complexité $O(n)$). On peut utiliser un dictionnaire, une table de hachage, ou un **arbre binaire de recherche équilibré** (type **arbre AVL**) pour P et Q :

- insertion et recherche dans un AVL requièrent seulement $O(\log n)$.

Il reste à associer une clé à chaque état ; en prolog, cle(u) = u et on compare les clés avec '@<='

II.3.4 Exemple : le jeu du takin.



Départ Arrivée

II.3.4.1 LE PROBLEME

Trouver une suite d'actions pour passer de Départ (u0) à Arrivée en un minimum de coups.

II.3.4.2 UNE MODELISATION

- chaque action revient à déplacer le "trou" (à **gauche**, à **droite**, vers le **haut**, vers le **bas**).
- k(u,v) = 1 $\forall v \in$ suc(u) : toutes les actions ont le même coût.
- g(u) = nb total de déplacements déjà effectués depuis le début
- h(u) = estimation du nb total de déplacements restants pour parvenir à la situation d'arrivée, ex : on prend **h_1(u) = nombre de pièces mal placées** par rapport à la situation d'arrivée (le trou n'est pas une pièce).

– h_1 est **monotone** . →Démonstration : en 1 action de u à $v \in S(u)$, de coût k(u,v) = 1,

soit cette action place correctement une pièce qui était mal placée, et on a : $h_1(v) = h_1(u) + 1$

soit cette action déplace une pièce qui était bien placée, et on a : $h_1(v) = h_1(u) - 1$

soit cette action est nulle (la pièce reste mal placée) : $h_1(v) = h_1(u)$

On peut donc affirmer quel $\forall v \in S(u)$ l'action vérifie : $-1 \leq h_1(u) - h_1(v) \leq 1$

Comme k(u,v) = 1, on peut donc en déduire que **$h_1(u) - h_1(v) \leq k(u,v)$**

– h_1 est **minorante** : **chaque action ne modifie la position que d'une seule pièce à la fois** ; donc s'il y a *n* pièces mal placées il faudra au moins *n* itérations (une pour chaque pièce) ; or pour une pièce donnée, il faut 1 ou plusieurs actions pour qu'elle arrive à destination. h est pourtant calculée comme si chaque pièce mal placée pouvait être bien placée en 1 seule action. C'est optimiste car cela ne tient aucun compte de la distance à parcourir ni des obstacles à contourner ; h minore le nombre de déplacements nécessaires.

– h est **coïncidente** : h(t) = 0 pout $\forall t \in T$: effectivement s'il n'y aucune pièce mal placée, c'est qu'on a atteint le but.

Règles de construction de l'arbre de recherche développé par A*

- la racine est u_0
- les états sont numérotés par **ordre croissant d'apparition** au cours de l'exécution de A* ;

  développement de u_0 $S(u_0) = \{u_1, u_2, u_3\}$, puis

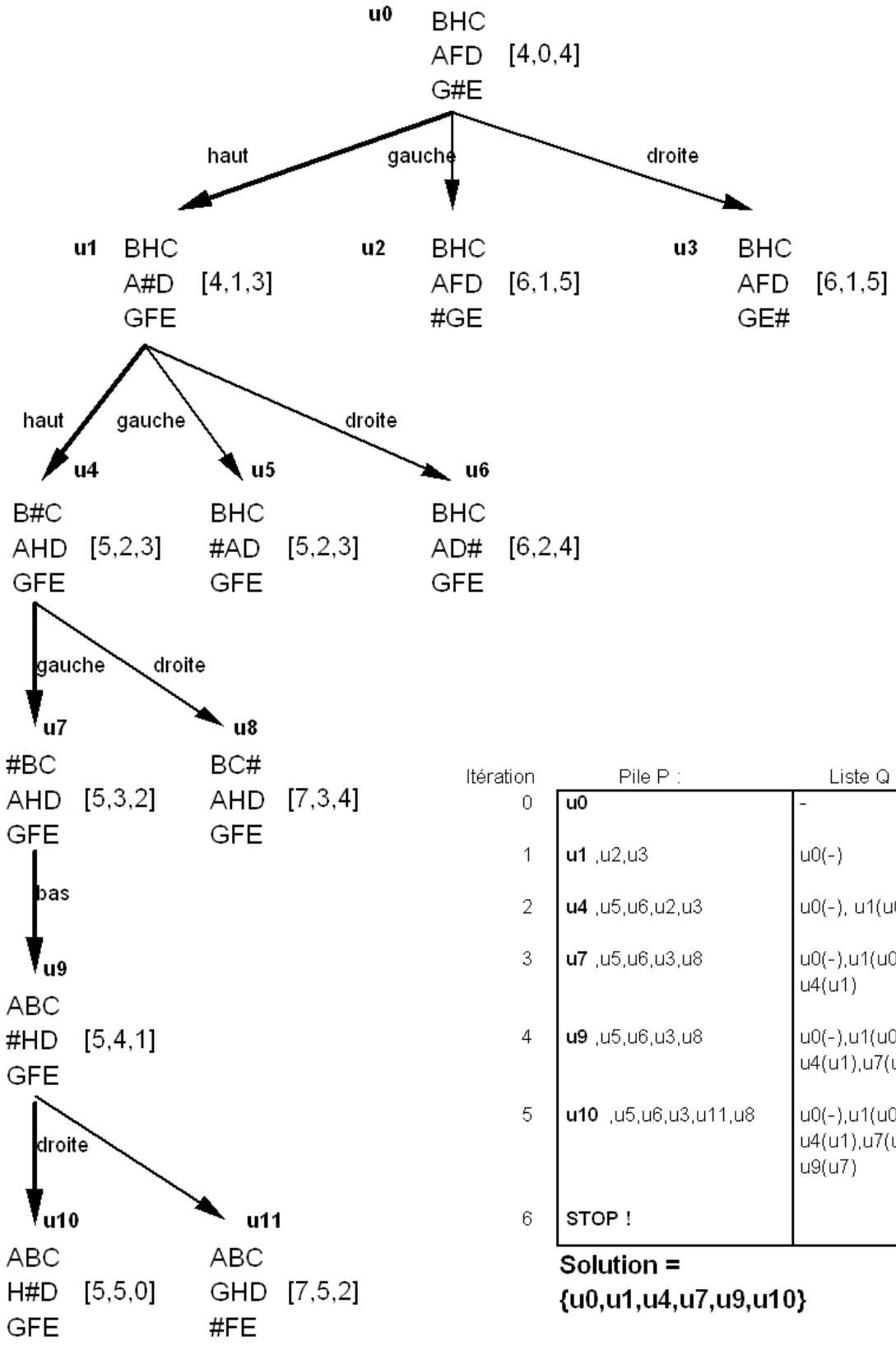
  développement de u_1 $S(u_1) = \{u_4, u_5, u_6\}$, puis

  développement de u_4, $S(u_4) = \{u_7, u_8\}$, etc

  Cette numérotation permet de retrouver les étapes de A* à partir de l'arbre.
- sur chaque arc (u,v), on fait figurer le nom de l'action et son coût k(u, v)
- en chaque nœud, on fait figurer son numéro d'apparition, une structure de données décrivant l'état, et les valeurs de [f,g,h].
- ex :

u_0 : $\begin{matrix} b & h & c \\ a & \ldots & d \\ g & f & e \end{matrix}$ [4 0 4]

–



Itération	Pile P :	Liste Q :
0	**u0**	-
1	**u1** ,u2,u3	u0(-)
2	**u4** ,u5,u6,u2,u3	u0(-), u1(u0)
3	**u7** ,u5,u6,u3,u8	u0(-),u1(u0), u4(u1)
4	**u9** ,u5,u6,u3,u8	u0(-),u1(u0), u4(u1),u7(u4)
5	**u10** ,u5,u6,u3,u11,u8	u0(-),u1(u0), u4(u1),u7(u4) u9(u7)
6	STOP !	

Solution = {u0,u1,u4,u7,u9,u10}

III Recherche heuristique dans les arbres de jeux

III.1 Définitions

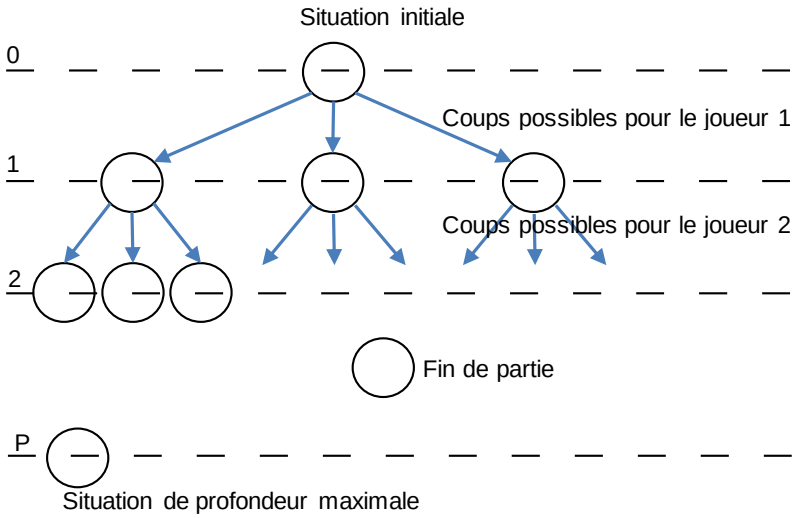
On considère des jeux à 2 joueurs :

- parfaitement informés : la totalité de la situation est connue de chaque joueur
- pas de hasard, toute situation résulte des décisions prises par les joueurs
- les 2 adversaires jouent alternativement
- arrêt du jeu lorsqu'un état "gagnant" pour l'un des joueurs est atteint (ou éventuellement 1 partie nulle)
- exemples : Tic-Tac-Toe (Morpion), Puissance4, Echecs, Dames, Othello, Awélé, Go ...

Déterminer l'**arbre complet** est irréaliste ; on ne développe qu'un **arbre partiel**, pour P coups à partir de la situation initiale. P = **profondeur** maximale d'une feuille.

Arbre de jeu

- 1 nœud = 1 situation du jeu
- 1 arc = 1 coup jouable (légal)
- racine = la situation initiale
- feuilles = situations non développées



Attention, 2 types de **feuilles** (nœuds que l'on n'a pas le droit de développer)

- des **situations** où on a atteint la **profondeur maximale**
- des **fins de partie** : un joueur gagne (l'autre perd) ou la partie est déclarée nulle.

III.2 Fonction d'évaluation des feuilles

On peut évaluer chaque feuille à l'aide d'une **fonction heuristique** $h(u, J)$ permettant d'évaluer le caractère favorable ou défavorable de la situation u pour le joueur J.

ex : aux Echecs, cette fonction peut prend en compte :

- le « poids » total des pièces restantes, le poids de celles occupant le "centre" de l'échiquier
- la possibilité de "roquer"
- la mobilité des pièces importantes (nb total de cases protégées, nb de cases atteignables par la reine, les fous, les cavaliers et les tours)
- nombre de pions encore capables d'aller sur la dernière ligne

La fonction $h(u,J)$ est en général **centrée** :

- $h(u,J) = 0$ pour une situation S neutre (ni favorable ni défavorable à J)
- $h(u,J) = -h(u, J')$ pour $J' = adversaire(J)$

Convention **Minmax** :

On évalue toutes les situations **du seul point de vue du 1er joueur** (celui qui joue à la racine).

La fonction $h(u)$ est dédiée au 1er joueur ; ex : aux Dames les blancs commencent et on évaluera une feuille u par : $h(u)$= nb de pions **blancs** en u – nb de pions **noirs** en u

Le **1er joueur est appelé max** (il cherche à maximiser $h(u)$) ; au contraire le 2ème joueur est appelé **min**, car cherche à minimiser $h(u)$.

- $h(u) = +\infty$ si **max** gagne en u ($\leftrightarrow$ min perd)
- $h(u) = -\infty$ si **min** gagne en u ($\leftrightarrow$ max perd)

Convention **Negamax** :

On évalue toute situation u selon le point de vue du joueur courant (celui ayant le tour en u)

La fonction $h(u,J)$ est une fonction d'évaluation pour le joueur courant J.

- $h(u,J) = +\infty$ si u est **gagnante pour J**
- $h(u,J) = -\infty$ si u est **perdante pour J**

Pb : disposant d'une fonction heuristique applicable aux feuilles de l'arbre, comment déterminer le meilleur coup à jouer par le premier joueur dans la situation initiale (à la racine) ?

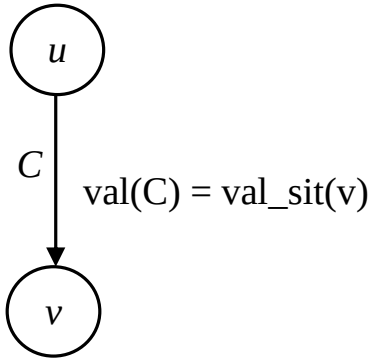
III.3 La procédure MinMax

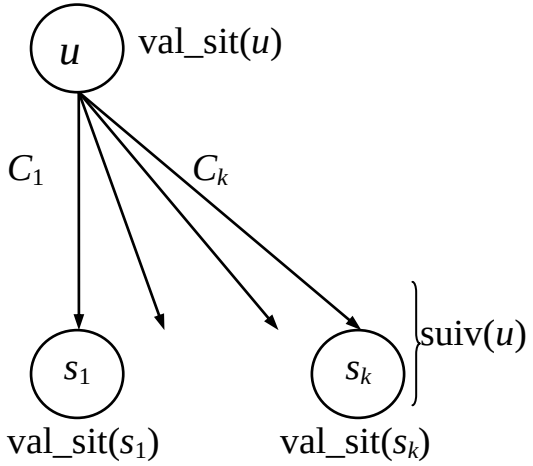
III.3.1 Principe

La fonction *h* ne s'applique qu'aux feuilles de l'arbre (situations non développables).

Pour les nœuds internes on doit évaluer l'intérêt de chaque coup jouable : la valeur d'un coup est celle de la situation atteinte par ce coup.

S'il y a plusieurs coups jouables à partir de *u*, c'est le meilleur qui donnera sa valeur à u.





Si u est une situation dans laquelle le 1er joueur (max) doit jouer : $\boxed{\text{val\_ sit}(u) = \max_{s \in \text{suiv}(u)} \text{val\_ sit}(s)}$

Si u est une situation dans laquelle le 2e joueur (min) doit jouer, $\boxed{\text{val\_ sit}(u) = \min_{s \in \text{suiv}(u)} \text{val\_ sit}(s)}$

Si u est terminale (feuille), alors on utilise $h(u)$: $\boxed{\text{val\_ sit}(u) = h(u)}$

III.3.2 Cas particuliers

III.3.2.1 Fins de partie

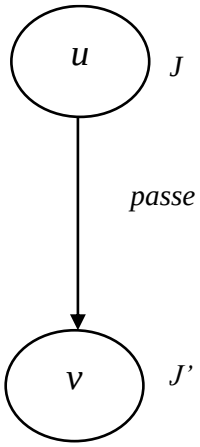
On a suiv(u)= $\varnothing$ et $\boxed{\text{val\_ sit}(u) = h(u) = -\infty}$ $\boxed{\text{val\_ sit}(u) = h(u) = +\infty}$ $\boxed{\text{val\_ sit}(u) = h(u) = 0}$

(perdant = max) (perdant = min) (partie nulle)

III.3.2.2 Cas des jeux dans lesquels on peut "passer son tour" (ex : Othello)

On accepte qu'un joueur puisse « passer » (ne joue rien) : la situation u ne change pas : $v = u$

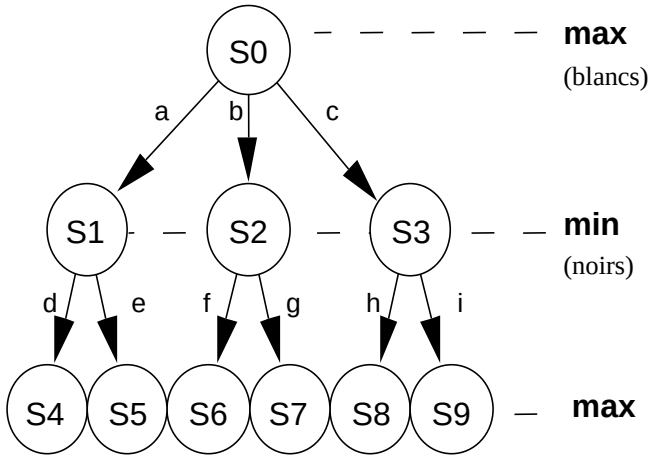
val_sit(u) = val_sit(v)



III.3.3 Application sur un exemple

ex : au jeu Othello, les blancs (=max) commencent, on peut utiliser la fonction heuristique suivante :

***h(S)* = nb de pions blancs – nb de pions noirs**



val_sit(S1) = min(2,7) = 2 = val_coup(a)

val_sit (S2) = min(1,8) = 1 = val_coup(b)

val_sit (S3) = min(1,6) = 1 = val_coup(c)

val_sit(S0) = max(2,1,1) = 2 = val_coup(a) **=> a est le meilleur coup** (pour une profondeur d'analyse de 2).

III.3.4 L'algorithme minmax

III.3.4.1 CONVENTION NEGAMAX

Les définitions données en IV.4.3 font implicitement intervenir le 1er joueur, max (celui qui joue à la racine de l'arbre de jeu). On évalue désormais une situation de façon relative, **du point de vue du joueur courant (qui joue en *u*)**. Pour une situation développable :

$$\text{val\_sit}(u, J) = \max_{s \in \text{suiv}(u)} (-\text{val\_sit}(s, J'))$$

avec J' = adversaire(J)

Comme max(-*f(x)*) = -min(*f(x)*) :

$$\text{val\_sit}(u, J) = -\min_{s \in \text{suiv}(u)} (\text{val\_sit}(s, J')) \qquad \text{(N)}$$

Si la profondeur maximale est atteinte en *u*, ou si *u* est une fin de partie :

$$\text{val\_sit}(u, J) = h(u, J)$$

D'autre part on modifie également la définition de l'heuristique $h(u,J)$ pour les fins de partie :

$h(u, J) = +\infty$	$h(u, J) = -\infty$	$h(u, J) = 0$
si J gagne en u	si J perd en u	si partie nulle

ex : pour Othello, une fonction heuristique naïve est la suivante :

$h(u, J)$ = nb de pions de J – nombre de pions de J'

On a donc pour une situation u :

$h(u, \text{noirs})$ = nb de pions noirs en u – nb de pions blancs en u

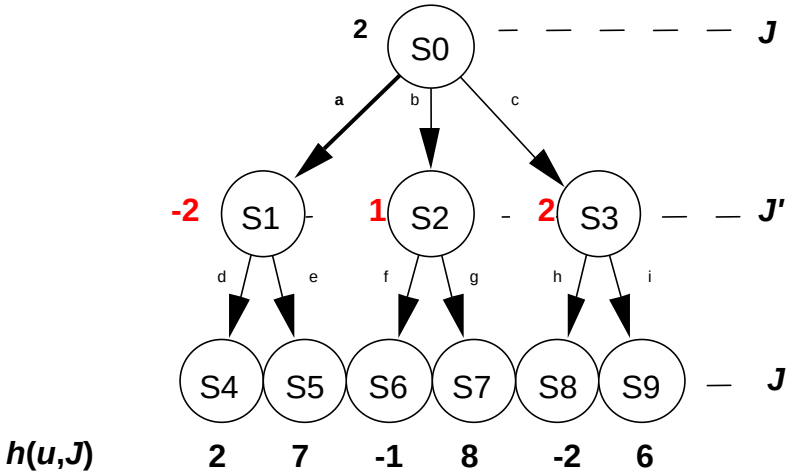
$h(u, \text{blancs})$ = nb de pions blancs en u – nb de pions noirs en u = – $h(u, \text{noirs})$

$$\boxed{h(u, J) = - h(u, \text{adv}(J))}$$

III.3.4.2 APPLICATION A L'EXEMPLE PRECEDENT

Le meilleur coup en S0 est bien le coup "a" (CQFD).

Avec Negamax, les nœuds intermédiaires (S1, S2, S3), et plus généralement les situations résultant d'un nombre impair de coups depuis la racine, ont une valeur opposée à celle obtenue avec minmax.



Dans la procédure qui suit :

J	joueur qui joue en *u*.
P	profondeur courante en *u* (nb de coups depuis la racine)
Pmax	profondeur maximale de recherche
*C**	meilleur coup que doit jouer *J* en *u*
*F**	évaluation de *u* = évaluation de *C**

On note :

↓A	A argument donné à l'appel (de type *in*)
↑A	A argument retourné (de type *out*)

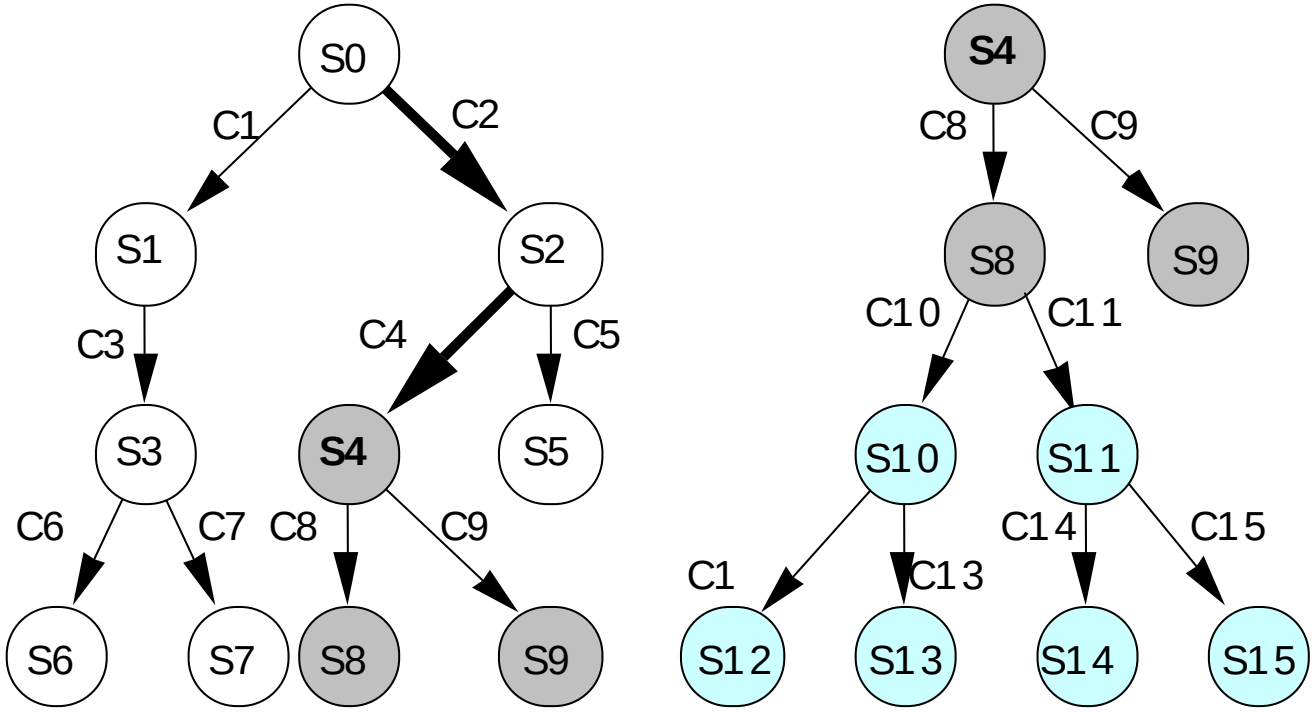
procedure **Negamax**(↓*J*, ↓*u*, ↓*P*, ↓*Pmax*, ↑*C**, ↑*F**)

```
début
    si P = Pmax ou si fin_de_partie(u) alors
        C* ← null                          // pas de coup jouable si profondeur maximale atteinte ou fin de partie ;
        F* ← h(u,J)                        // on utilise alors la fonction heuristique pour évaluer la situation
    sinon  Coups ← {C | J peut jouer C en u }
        si Coups = ∅ alors                 // la situation n'est pas une fin de partie et on peut « passer »
            C* ← passe                     // J doit "passer son tour"
            Negamax(adv(J), u, P+1, Pmax, C0, F0)
            F* ← F0
        sinon
            J' ← adv(J)
            i* ← 1
            pour i ∈ 1..|Coups| faire
                s[i] ← jouer(J, Coups[i], u)
                Negamax(J', s[i], P+1, Pmax, C[i], F[i])
                si F[i] < F[i*] alors i* ← i          // mise à jour du coup qui amène au minimum
                finsi
            finpour
            C* ← Coups[i*]   F* ← - F[i*]             // c'est le coup qui amène en s[i*] qui est intéressant
        finsi                                         // ne pas oublier que max(-f(x)) = -min(f(x))
    finsi
fin
```

III.3.5 Algorithme pour joueur automatique

A l'aide de l'algo Minmax, le programme décide du meilleur coup à jouer en *u* (résultat = situation *u'*), puis attend (lit) la riposte de l'adversaire (résultat = situation *u''*).

Il reconstruit un nouvel arbre partiel issu de *u''* avec la même profondeur (voir fig. ci-dessous).



arbre précédant le coup C2 — arbre suivant (après C2 et C4)

```
Procedure Joueur_automatique(Pmax)
Début                                          // Initialisations
    C      ← init                              // coup fictif précédant la racine
    u      ← situation-initiale
    J      ← tirage_au_sort                    // parmi {ordinateur, utilisateur}
    P      ← 0
    si     J = utilisateur alors               // si l'utilisateur commence la partie
           lire(C_utilisateur)                 // on lit le coup
           u ← jouer(C_utilisateur, u)         // et on le joue
    finsi
    tantque non fin_de_partie(u)               // Boucle principale du jeu
           Negamax(ordinateur, u, P, Pmax, C*, F*)
           u ← jouer(C*, u)                    // jouer le meilleur coup fourni par Negamax
           si non fin_de_partie(u) alors
                  lire(C_utilisateur)          // lire la riposte de l'utilisateur
                  u ← jouer(C_utilisateur, u)
           finsi
    fintantque
    cas  F* =                                  // Les 3 fins de parties possibles
           -∝    alors échec                   // l'ordinateur perd (l'utilisateur gagne)
           + ∝   alors succès                  // l'ordinateur gagne (l'utilisateur perd)
           0     alors partie nulle            // la partie est nulle
    fin cas
```

III.3.6 Exemple du TicTacToe (morpion 3x3)

Aligner N symboles identiques (× ou o sur une même ligne, colonne, ou diagonale, dans une grille carrée N×N, initialement vide.

On se limite au cas N=3.

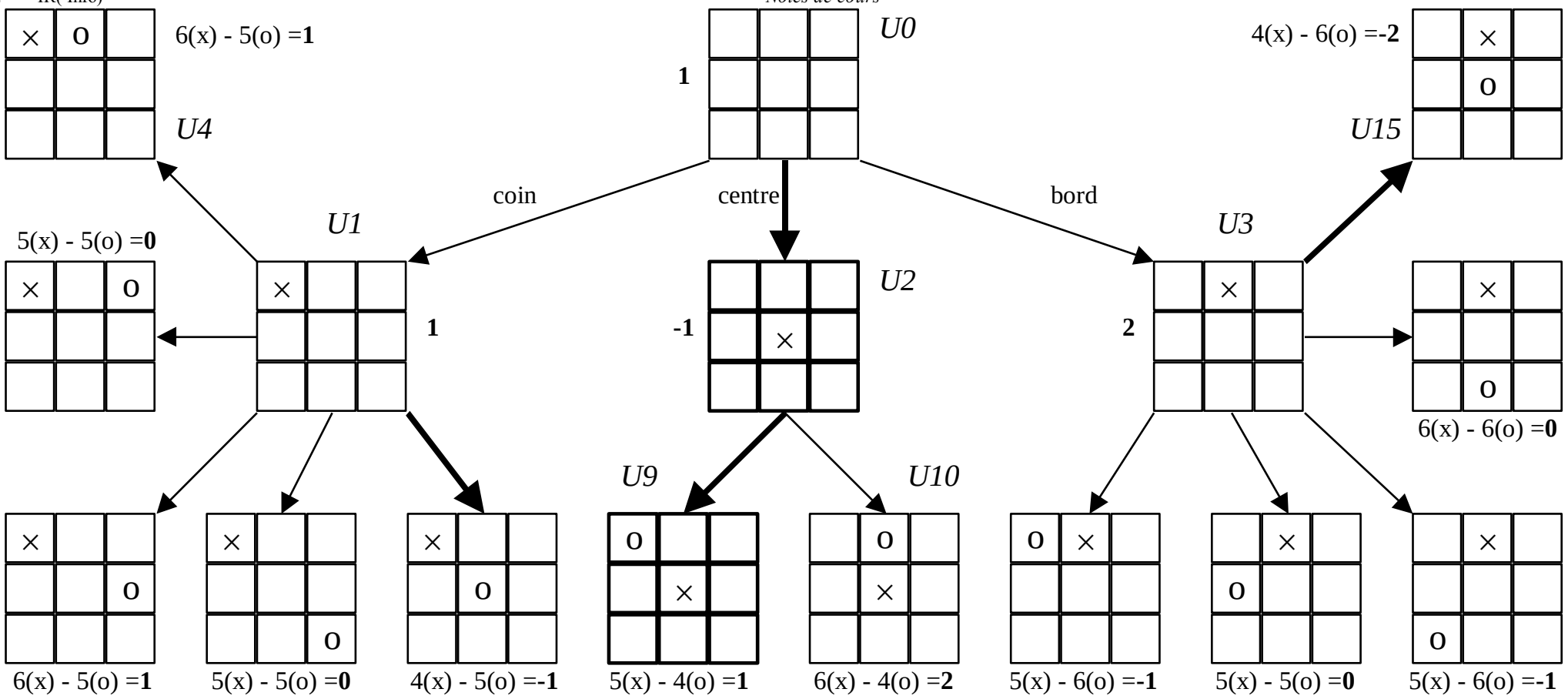
On tire au sort le premier joueur : (×) commence.

Avant de jouer, celui-ci effectue une analyse Negamax de profondeur 2.

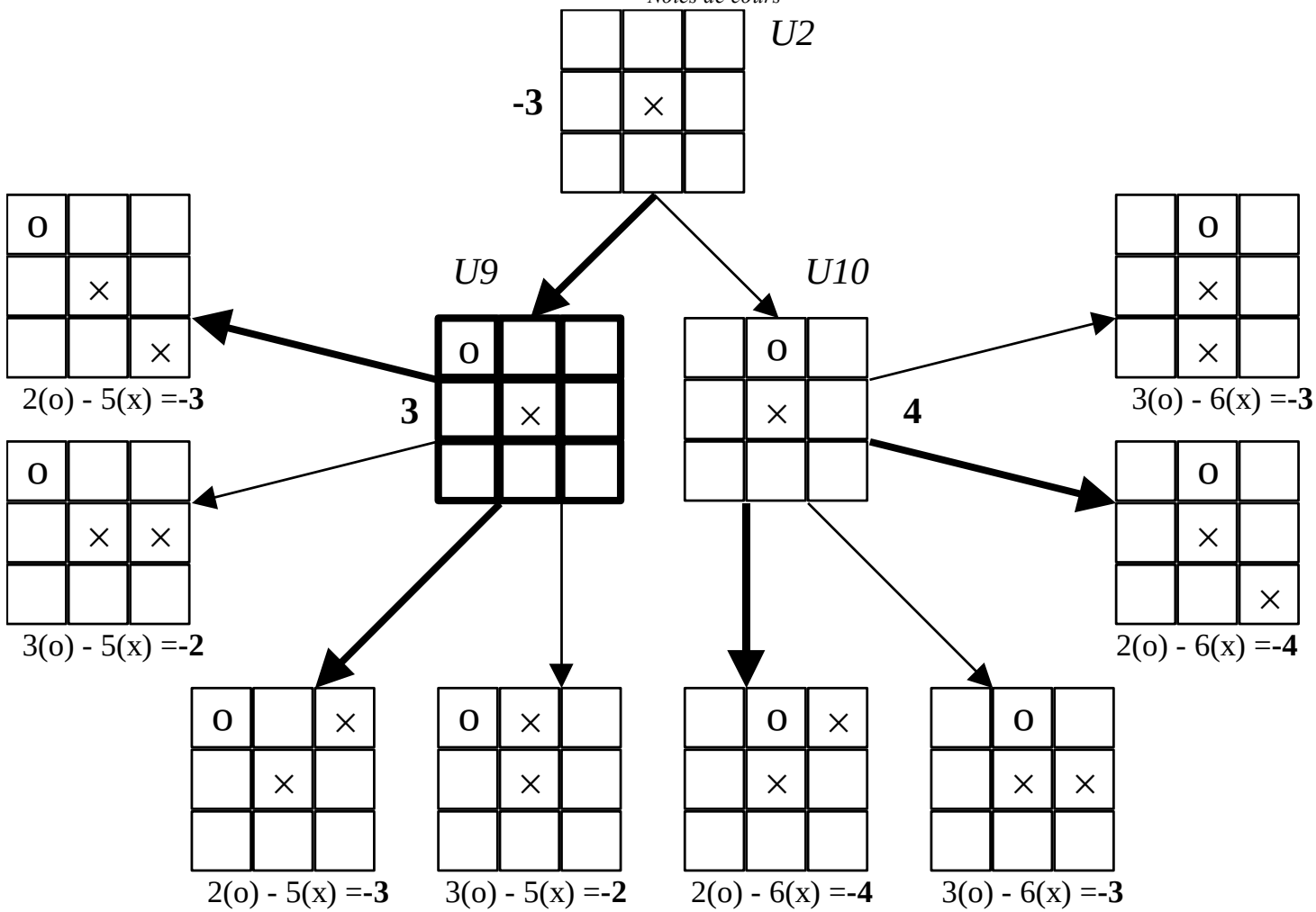
Une heuristique possible :

On évalue la **différence entre le nombre d'alignements encore possibles pour *J* et le nombre d'alignements encore possibles pour l'adversaire**. Un alignement est déclaré ***possible*** sur une ligne (resp. colonne ou diagonale) tant que l'adversaire n'occupe aucune case de cette ligne (resp. colonne ou diagonale) :

$h(u,J)$ = **nb total de (lignes, colonnes et diagonales) ne contenant aucun symbole de adv(*J*)**
- nb total de (lignes, colonnes et diagonales) ne contenant aucun symbole de *J*



Evaluation du meilleur coup initial pour les '×'

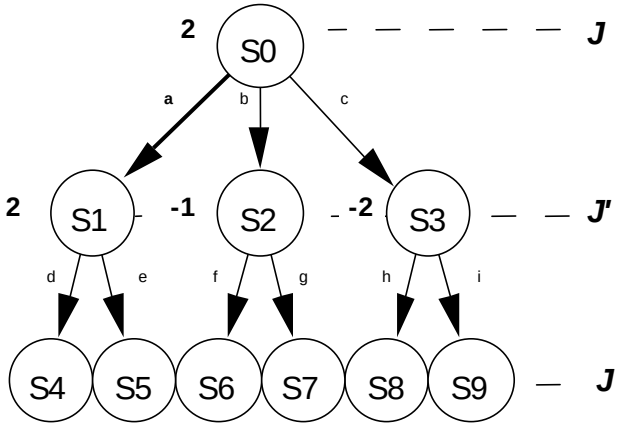


Evaluation de la meilleure riposte pour les 'o'

III.4 La procédure αlpha-βeta

III.4.1 Critique du MinMax

Etant donné une situation *u* et une profondeur *P* données, **tout l'arbre partiel** issu de *u* est exploré et évalué. Or, compte tenu de l'ordre de parcours (profondeur d'abord) de cet arbre durant l'évaluation Minmax, il existe des situations qu'il est inutile de développer, exemple :



situation *u*	SO →	S1 →	S4 →	S1 →	S5 →	S1 →	S0
val(*u*)	?	?	=2	≤2	=7	=2	≥2

A ce stade, on est sûr que l'évaluation de S0 ne peut pas être < 2 (puisque on peut assurer la valeur 2 en jouant "a"). On peut arrêter l'évaluation fine de S2 dès qu'on peut prouver que val(S2) ≤ 2. Idem pour S3.

Retour à l'exemple : suite de l'exploration :

situation	S0 →	S2 →	S6 →	S2 →	~~S7~~
val(*u*)	≥2	?	=-1	≤-1	~~=7~~

Stop : on sait que la valeur de S2 sera inférieure ou égale à -1 (car c'est un nœud de type min), quelle que soit la valeur exacte de la situation S7 et que cette valeur est < à la valeur 2 du coup a.

On peut assurer la valeur +2 en S1 en jouant "a" => il est inutile de continuer à évaluer finement les suivants de S2. On remonte donc directement en S0.

Suite de l'exploration :

situation	SO →	S3 →	S8 →	S3 →	~~S9~~
val(*u*)	≥2	?	=-2	≤-2	~~=6~~

Stop : il est inutile d'explorer S8 pour les mêmes raisons que précédemment ; on sait que S3 aura une valeur limitée supérieurement à -2. Or on peut assurer la valeur +2 en S1 en jouant "a" => il est inutile de continuer à évaluer les suivants de S3. On remonte en S0 et l'exploration est terminée.

Bilan : 2 branches ont été élaguées.

III.4.2 Principe d'élagage - Définitions

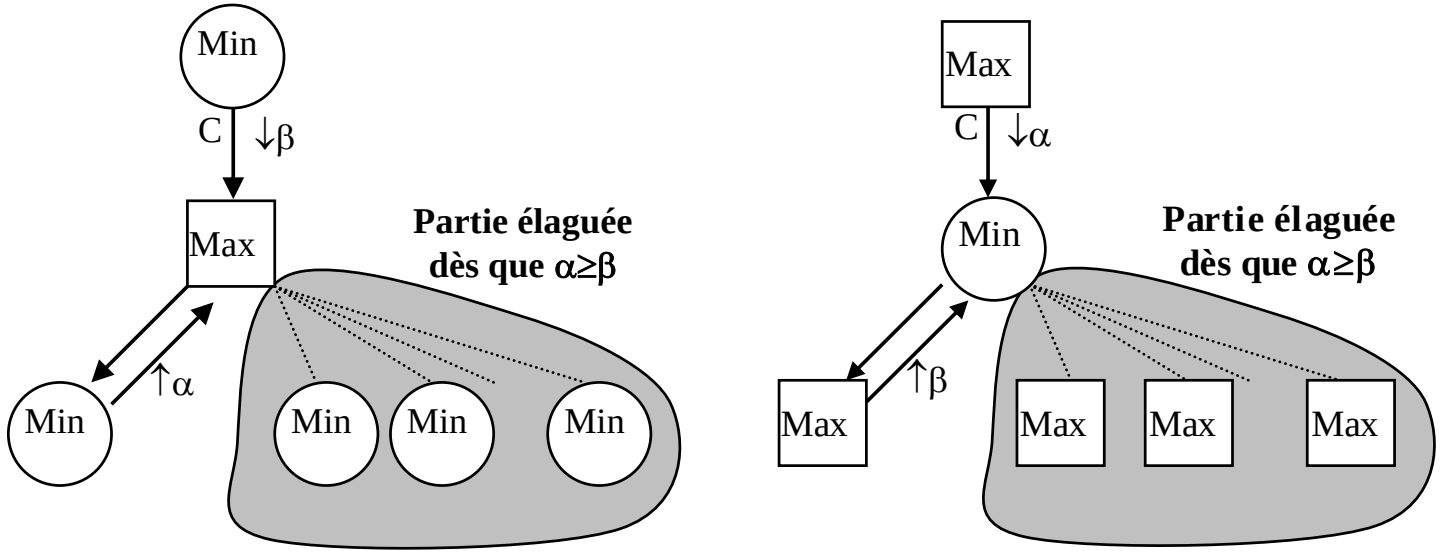
On appelle α-valeur d'un nœud de type Max une borne minimale courante de la valeur de ce nœud, après exploration d'un certain nombre des nœuds suivants. Si aucun suivant n'a encore été exploré $\alpha = -\infty$

On appelle β-valeur d'un nœud de type Min, une borne maximale courante de la valeur de ce nœud, après exploration d'un certain nombre des nœuds suivants. Si aucun suivant n'a encore été exploré $\beta = +\infty$

En une feuille de l'arbre on a : $\alpha = \beta = \text{val}(u) = h(u)$

De plus pendant l'exploration, lorsqu'on descend d'un nœud Max (resp. Min) vers un nœud Min (resp. Max), la valeur α (resp. β) du nœud père est transmise au nœud fils.

L'exploration des fils Min (resp. Max) d'un nœud Max (resp. Min) *u* s'arrête dès que la valeur α (resp. β) qu'il reçoit d'un de ses fils dépasse la valeur β (resp. α) qu'il a reçu de son père.



III.4.3 AlphaBeta

Pour alléger, on limite l'algorithme au seul calcul de la valeur val(*u*). Il restera à identifier le coup qui retourne cette valeur en *u*. Dans un premier temps on n'utilise pas la convention Negamax.

Le premier appel s'effectue avec A(lpha) = `-INFINI`, B(eta) = `+INFINI`.

```
fonction ALPHABETA(J, U, P, Pmax, A, B)
debut
    si P = Pmax ou fin_de_partie(u) alors
        retourner h(u)
    sinon
        Coups ← Coups_Jouables(J, U)
        i ← 1
        Elagage ← Faux
        si J = max      alors
            Meilleur  ← -INFINI
            tantque i ≤|Coups| et non Elagage
                s ← jouer(J,Coup[i],U)              -- Coup[i] amène en s
                Val ← ALPHABETA(adv(J),s,P+1,Pmax,A,B)
                si Val > Meilleur alors
                    Meilleur ← Val                  -- actualisation de Meilleur
                    si Meilleur > A alors
                        A ← Meilleur                -- actualisation de Alpha
                        si A > B alors              -- élagage (sortie de boucle)
                        Elagage ← Vrai
                        finsi
                    finsi
                finsi
            fintanque
```

```
        sinon                   -- J = min
            Meilleur ← +INFINI
            tantque i ≤|Coups| et non Elagage
                s ← jouer(J,Coup[i],U)          -- Coup[i] amène en s
                Val ← ALPHABETA(adv(J),s,P+1,Pmax,A,B)
                si Val < Meilleur alors
                    Meilleur ← Val              -- actualisation de Meilleur
                    si Meilleur < B alors
                        B ← Meilleur            -- actualisation de Beta
                        si A > B alors          -- élagage (sortie de boucle)
                            Elagage ← Vrai
                        finsi
                    finsi
                finsi
            fintantque
        finsi
        retourner Meilleur
    finsi
fin
```

Version avec convention negamax :

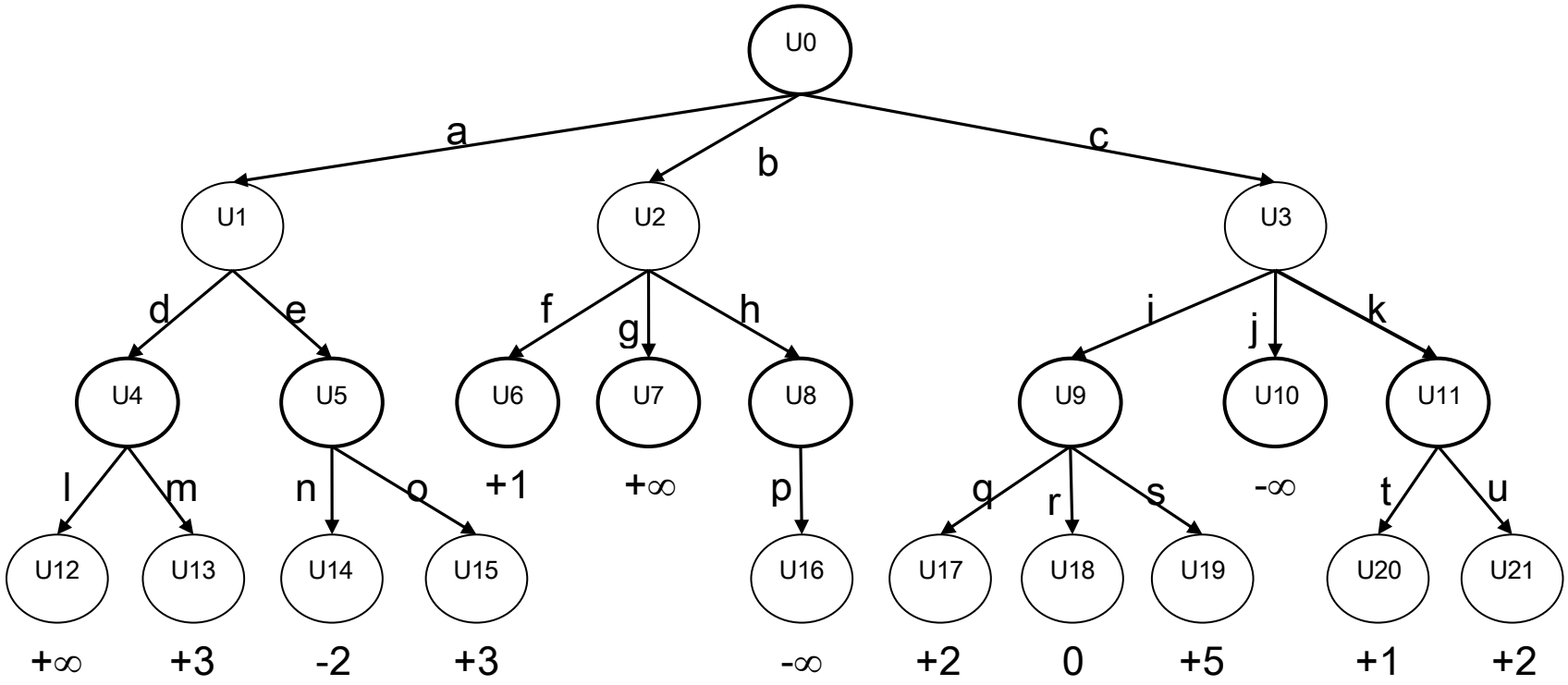
```
fonction ALPHABETA(J,U,P,Pmax,  A, B)
début
  si P = Pmax ou si fin_de_partie(U)      -- U est une feuille
  alors
     retourner h(U,J)
  sinon
     Meilleur  ← –INFINI
     Elagage   ← Faux
     Coups     ← Coups_Jouables(J,U)
     i         ← 1
     tantque i ≤ |Coups| et non Elagage
        s ← jouer(J,Coup[i],U)          -- Coup[i] amène en s -- on cherche l'opposé des valeurs suivantes
        Val ← - ALPHABETA(adv(J),s,P+1,Pmax, -B, -A)
        si Val > Meilleur alors
           Meilleur ← Val                -- actualisation de Meilleur
           si Meilleur > A alors
              A ← Meilleur               -- actualisation de A(lpha)
              si A > B alors             -- élagage (sortie de boucle)
                 Elagage ← Vrai
              finsi
           finsi
        finsi
     fintantque
     retourner Meilleur
  finsi
fin
```

Exercice de TD

L'arbre de jeu ci-après a les caractéristiques suivantes :

profondeur max = 3 coups,

pas de convention negamax



1/ On désire appliquer l'algorithme minmax.

a. préciser la signification des valeurs -∞ et +∞ dans certains nœuds.

b. établir la valeur minmax de la situation initiale U0 et le meilleur coup à jouer pour le 1er joueur.

2/ On désire appliquer la convention negamax :

a. modifier les valeurs de l'heuristique h(S) pour obtenir h(S, J) = heuristique pour le joueur courant (celui qui joue en S)

b. modifier les valeurs de l'évaluation minmax en chaque nœud

c. vérifier que le meilleur coup en U0 est le même.

3/ On désire appliquer l'algorithme alpha-beta sans convention negamax

Indiquer quelles sont les branches de l'arbre qui sont élaguées par l'algorithme alpha-beta.

IV Recherche heuristique dans les graphes de sous-problèmes

IV.1 Formulation générale

Un graphe de sous-problèmes représente **la décomposition d'un problème** en sous-problèmes :

– la **racine** (nœud sans prédécesseur) représente le problème à décomposer

– les **feuilles** représentent des **problèmes résolus** (ou triviaux).

Différence avec les graphes d'états :

– un connecteur (règle) relie 1 nœud à k autres noeuds ; on parle de **k–connecteur**.

– la **solution** est un **sous-graphe**.

IV.2 Graphe ET/OU et hyper-graphes

Un **graphe ET/OU** est un graphe de sous-problèmes G(X,U) dans lequel X regroupe 2 types de nœuds :

– un **nœud ET** représente un problème décomposable en une **conjonction** de sous–problèmes.

– un **nœud OU** représente un problème décomposable en une **disjonction** de sous–problèmes.

IV.2.1 Exemple

Soit le système de production :

R1 : b,c → a

R2 : a,e,f → **d**

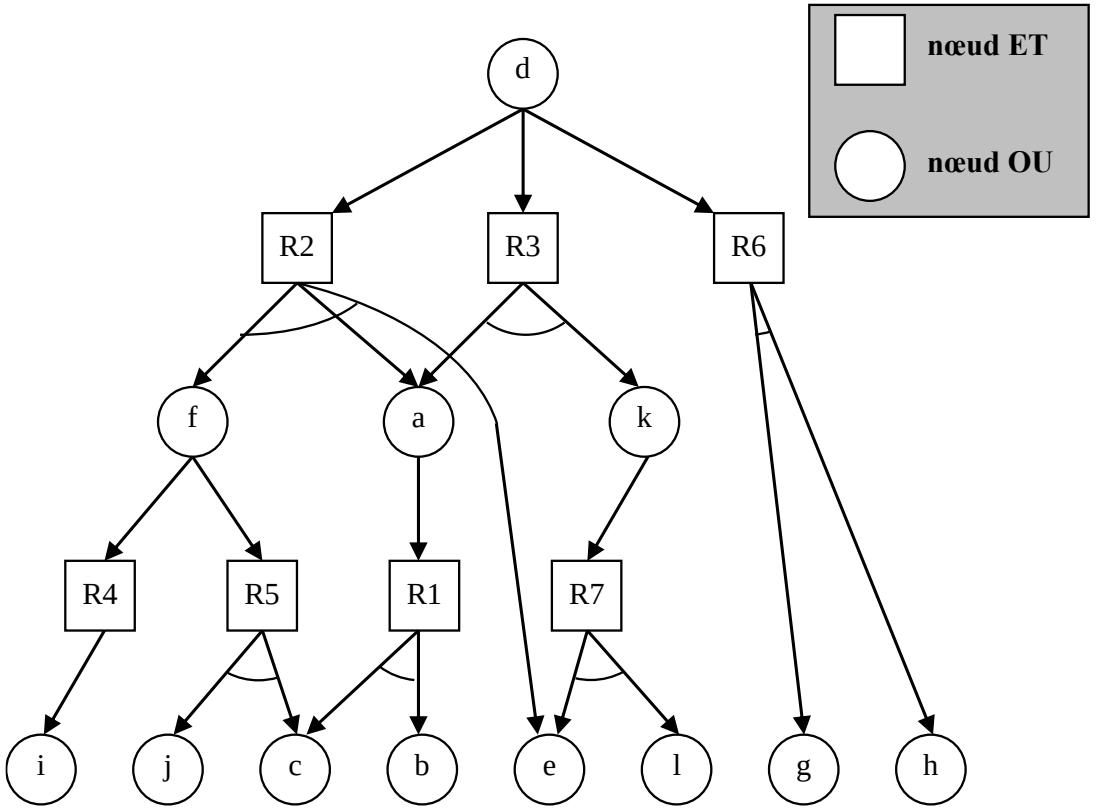
R3 : a,k → **d**

R4 : i → f

R5 : c,j → f

R6 : g,h → **d**

R7 : e,l → k

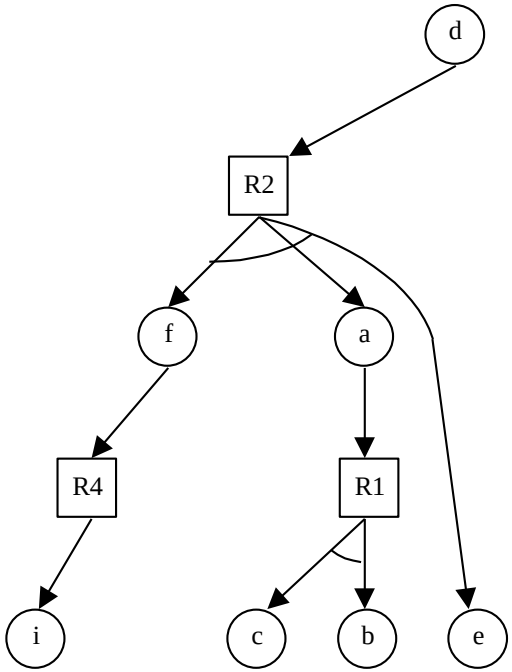


Un graphe ET/OU

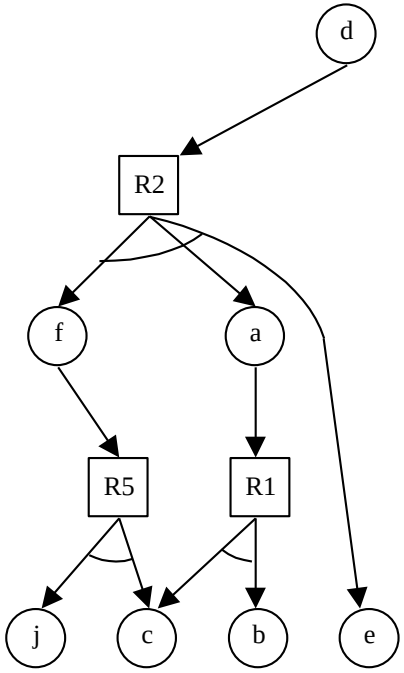
IV.2.2 Sous-graphes solutions

Un sous-graphe solution est donc un graphe qui comporte :

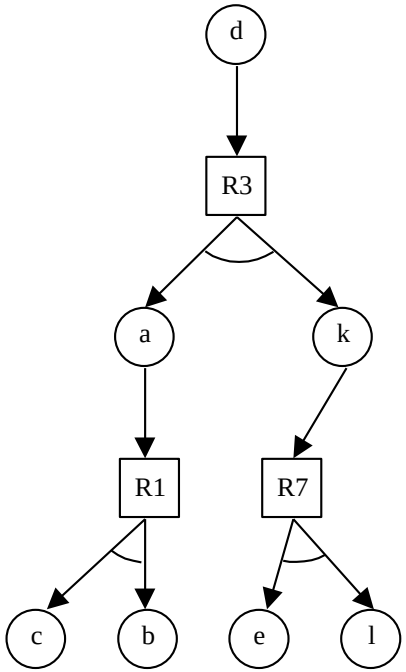
- la racine du graphe initial
- si un nœud ET appartient à cette solution, tous ses nœuds fils appartiennent aussi à la solution,
- si un nœud OU appartient à cette solution, un de ses nœuds fils appartient aussi à la solution.



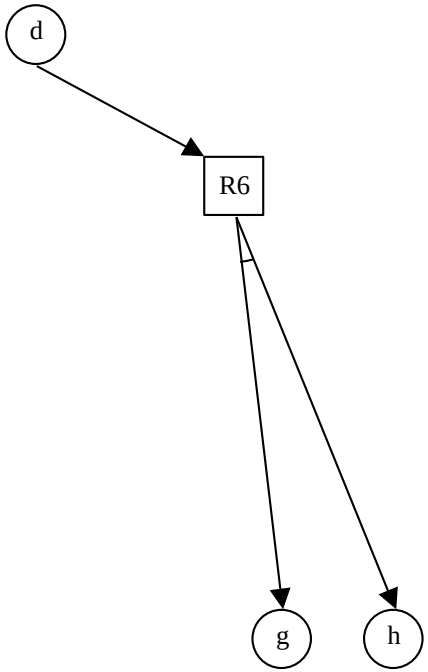
solution n°1



solution n° 2



solution n°3



solution n° 4

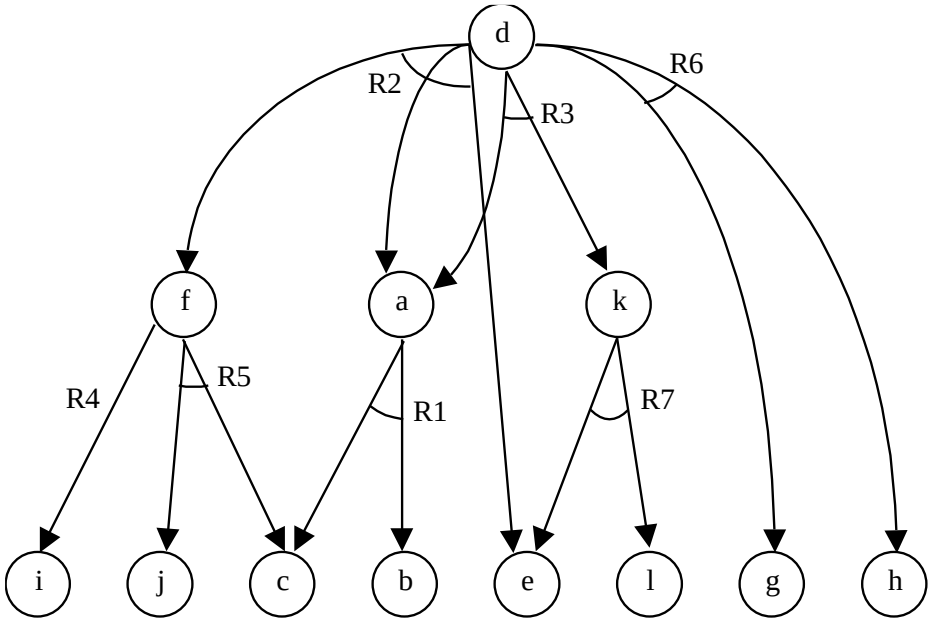
Quel est la solution la plus simple ? Une définition possible : celle qui nécessite le moins de faits (feuilles).

Solution = **base de faits minimale** permettant de résoudre le problème situé à la racine. Pour *d*, les bases de faits associées à chaque sous-graphe solution sont : {{*i*, *c*, *b*, *e*}, {*j*, *c*, *b*, *e*}, {*c*, *b*, *e*, *l*}, {*g*, *h*}}.

Plus généralement, on s'intéresse aux sous-graphes de coût total minimal. Comment définir le coût ?

IV.3 Recherche heuristique dans un hypergraphe

IV.3.1 Hypergraphe associé à un graphe et/ou



Hypergraphe associé au système de production

Pour passer d'un graphe ET/OU à un hyper-graphe, il suffit de transformer les nœuds ET en k-connecteurs, ex : le nœud ET 'R2' qui lie d à {f, a, e}, devient le 3-connecteur "R2".

IV.3.2 Algorithme AO*

On associe désormais un **coût à chaque k-connecteur**. Le problème est de déterminer le sous-graphe solution de coût total minimum.

notation :

G : hypergraphe

G' : un sous-graphe solution de G

u_0 : problème initial (racine)

T : problèmes résolus (feuilles de G)

R(u) : règles (ou k-connecteurs) permettant de décomposer u

S(u, r) : successeurs de u par la règle $r \in R(u)$

$\Omega(u)$: ensemble de tous les successeurs de u = $\bigcup_{r \in R(u)} S(u, r)$

c(u, r) : coût de la règle r appliquée à u.

On suppose que **l'hyper-graphe est acyclique** : tout nœud v qui est descendant d'un nœud u ne peut pas être également un ancêtre de u.

Le coût d'un sous-graphe solution de racine u et de feuilles N est défini récursivement :

si u ∈ N alors
k(u, N) = 0
sinon
k(u, N) = c(u,r) + k(s_1,N) + k(s_2,N) +… +k(s_i,N)

Pour éviter d'explorer tout le graphe, on utilise une fonction heuristique h(u) qui estime le coût de u avant de l'avoir développé. Après développement de u selon tous les connecteurs possibles, on actualise cette estimation en tenant compte du sous-graphe issu de u qui retourne l'estimation minimale. Si h est minorante, et si une solution est trouvée sans avoir tout développé tous les nœuds, on est sûr que l'exploration des nœuds non développés ne peut donner que des solutions de coût supérieur, et on peut arrêter l'exploration.

Soit h(u) une fonction heuristique estimant le coût du sous-graphe solution de racine u et h*(u) l'heuristique parfaite (inconnue).

On s'intéresse aux fonctions h(u) minorantes : $h(u) \leq h^*(u)$

Comme dans le cas de A*, on peut montrer que toute fonction **coïncidente** :

$h(n) = 0$ pour tout nœud terminal

et **monotone** :

$h(u) \leq c(u, r) + h(n_1) + h(n_2) + \ldots h(n_p)$ $\quad \forall r$ applicable à u et $S(u, r) = \{ n_1, n_2, \ldots, n_p\}$

est **minorante**.

Principes de l'algorithme AO*

Choix du nœud à développer : en priorité celui dont la fonction d'évaluation $q(u)$ est minimale.

- avant expansion d'un nœud :

$$\boxed{q(u) = h(u)}$$

- après expansion d'un nœud u et génération des successeurs de u

$$\boxed{q(u) = \min_{r \in R(u)} \left(c(u,r) + q\left(n_{r,1}\right) + \cdots + q\left(n_{r,k}\right) \right)}$$

où (notation) :

$R(u)$ est l'ensemble des connecteurs issus du noeud u (règles applicables en u) et

$\{n_{r,1}, n_{r,2}, \ldots n_{r,k}\}$ est l'ensemble des k successeurs de u après application du k-connecteur r

*Procedure AO**

// Initialisation

- G ← {u_0} (problème à résoudre).
- $q(u_0) \leftarrow h(u_0)$
- si est un problème terminal, marquer u_0 resolu

//Boucle principale en 2 phases : exploration descendante + révision des coûts ascendante

tantque u_0 n'est pas marqué resolu

// expansion d'un nœud (exploration descendante)

calculer G' , sous-graphe solution partiel en suivant tous les connecteurs **marqués** de G depuis u_0
sélectionner un nœud non développé (feuille) *n* de G'
déterminer R($\underline{n}$) et tous les successeurs de *n*
pour chaque successeur n_j de *n*
    si n_j n'est pas déjà dans G **alors**
        $q(n_j) \leftarrow h(n_j)$
        si n_j est un problème terminal alors resolu(n_j) ← VRAI
        fsi
    fsi
fp

// révision des coûts (exploration ascendante)

$S \leftarrow \{n\}$
tantque $S \neq \varnothing$
    enlever un nœud m de S tel que m n'a pas de descendants dans G qui apparaissent dans S
    pour chaque connecteur r liant m à des successeurs $\{n_{r1}, n_{r2}, \ldots, n_{rk}\}$)
        calculer $\boxed{q(m) = \min_{r \in R(u)} \left(c(u,r) + q(n_{r1}) + \ldots + q(n_{rk})\right)}$
        marquer le connecteur r qui permet d'obtenir ce minimum
        démarquer éventuellement le connecteur précédemment marqué issu de m
        si tous les successeurs $\{n_{r1}, n_{r2}, \ldots, n_{rk}\}$ sont marqués résolus **alors**
            resolu(m) $\leftarrow$ VRAI
        fsi
    fp
    si m est devenu résolu ou si $q(m)$ a évolué **alors**
        $S \leftarrow S \cup \{p\}$ tel que p est parent de m par un connecteur marqué
    fsi
ftq
ftq

Exemple

	n_0	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8
h	7	7	6	6	2	3	2	0	1