

Intelligence Artificielle

Méthodes de Résolution de Problèmes

Patrick Esquirol
INSA Toulouse

version du 29/01/2024

I Introduction

IA symbolique : représentation de connaissances, approche logique :

un système intelligent est capable de raisonner, de décider, de différencier la forme et le sens

IA numérique : apprentissage, approche statistique :

un système intelligent est capable d'apprendre par l'exemple, d'imiter

Le thème "Résolution de Problèmes" se situe plutôt dans le domaine de l'IA symbolique :

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

Application : problèmes combinatoires, robotique, productique, transport, démonstration, 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 : Echecs, 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

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

Il faut trouver un(des) circuit(s) pour livrer des articles ou collecter tous les articles retournés, en minimisant la distance totale parcourue, l'énergie dépensée, le temps de parcours ...

Algorithmes d'IA « classiques » :

- 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 parfois 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.

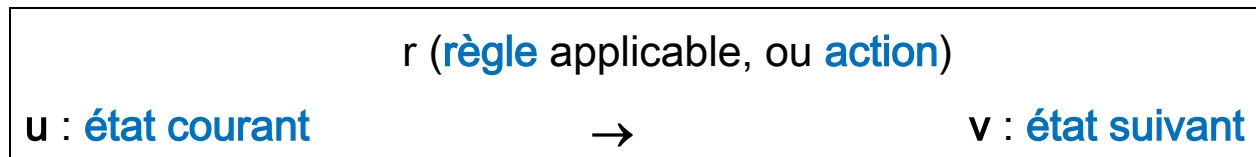
II Recherche heuristique dans les graphes d'états

II.1 Formulation générale

Soit :

- U = **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 = **état initial**
- T = 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 :



notation : $v = \text{suc}(r, u)$ (notation fonctionnelle)

$\text{transition}(r, u, v)$ (notation logique)

$k(u,v) = \text{coût}$ 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
 \Leftrightarrow **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**.

Il peut exister des **contraintes** sur les états et/ou sur les actions applicables :

états **admissibles** (vs états interdits)

règles/actions **légaux**.

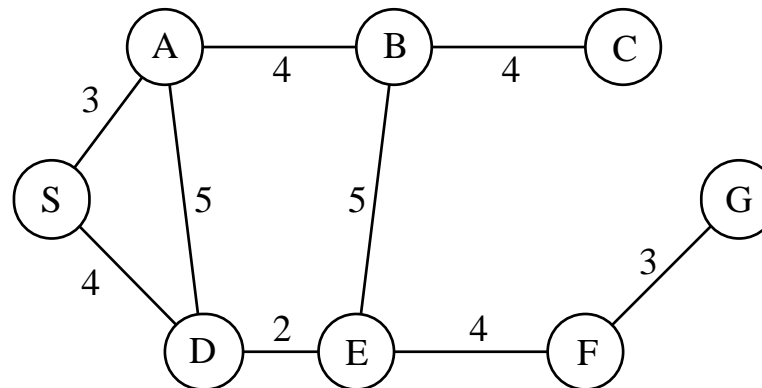
On note :

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

Représentation du graphe d'états, ex :

problème = rechercher le chemin le plus « court » entre S et G dans le réseau ci-dessous :

:



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.**

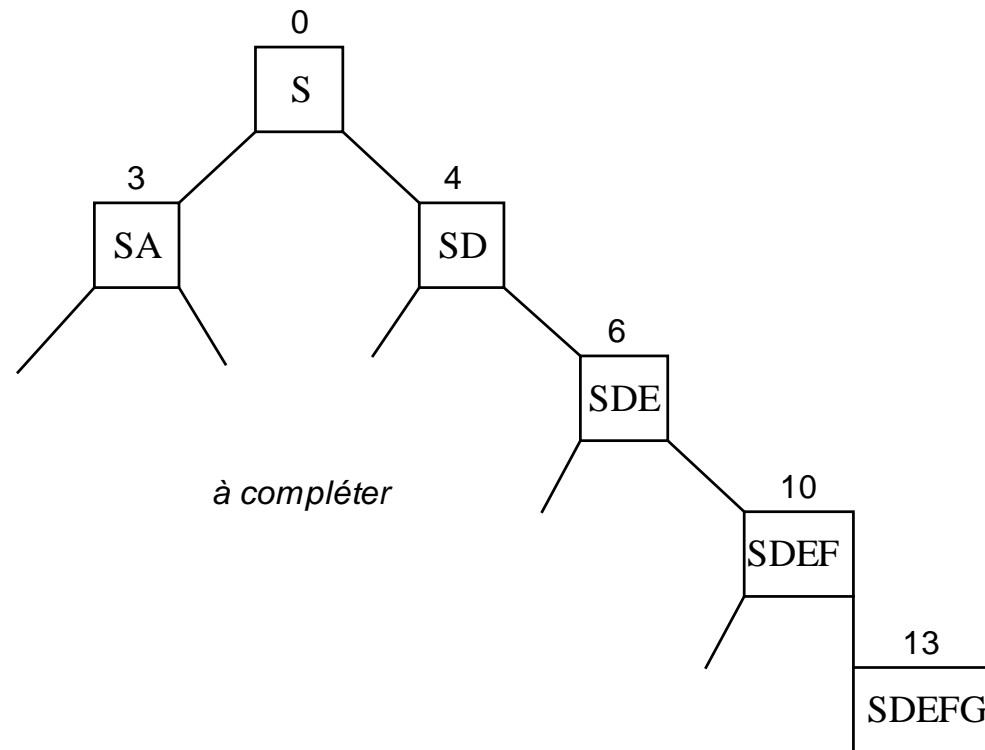
Coût entre 2 états u et v = *longueur [temps, consommation]* pour aller de u à v.

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

Calcul du **coût cumulé** dans un état u (appelé fonction « $g(u)$ » dans l'algorithme A^*):

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 réseau peut être grand (ex : Carte Europe) mais il est connu à l'avance. Les algorithmes qui calculent le plus court chemin (Bellman ou Dijkstra) peuvent être **matriciels**.
- en I.A., le **graphe des états d'un système** n'est pas donné à l'avance, on se donne seulement des règles de transition entre états ; l'espace n'est pas connu à l'avance, il 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 Algorithme de départ (sans prise en compte des coûts)

Il n'y a pas de critère d'optimisation. On cherche simplement à atteindre un état terminal admissible.

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 : Etat = {X,Y,Z,O, E, Items} avec :

X,Y,Z = variables de positions 3D, O = 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  $P \neq \emptyset$  et  $\text{sommet}(P) \notin T$ 
     $u_i \leftarrow \text{sommet}(P)$ 
    dépiler(P)
    déterminer  $S(u_i)$ 
    pour chaque  $s \in S(u_i)$ 
        empiler( $s$ , P)
    finpour
fintantque
3 si  $P \neq \emptyset$  alors  $\text{sommet}(P)$  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/

Problèmes des algos « de base » 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

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**.

Lorsqu'on développe l'extrémité du chemin partiel $C=\{u_0, u_1, \dots, u_i\}$ on interdit les états $u_{i+1} \in \{u_0, u_1, \dots, u_{i-1}\}$

Notation :

P = pile de chemins partiels non encore développés. On développe systématiquement le premier chemin partiel situé en haut de la pile **P** : $\text{sommet}(\mathbf{P})$.

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

$u_i = \text{dernier}(C) = \text{dernier}(\text{sommet}(\mathbf{P}))$ 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 $P \neq \emptyset$ et dernier(sommet(P)) $\notin T$

$C \leftarrow$ sommet(P) $u_i \leftarrow$ dernier (C)

 dépiler(P)

 déterminer $S(u_i)$

pour chaque $s \in S(u_i)$

si $s \notin \{u_0, u_1, \dots, u_{i-1}\}$

/ il faut éviter de boucler sur un état déjà atteint*/*

alors $C' \leftarrow \{u_0, u_1, \dots, u_i, s\}$

 empiler(C', P)

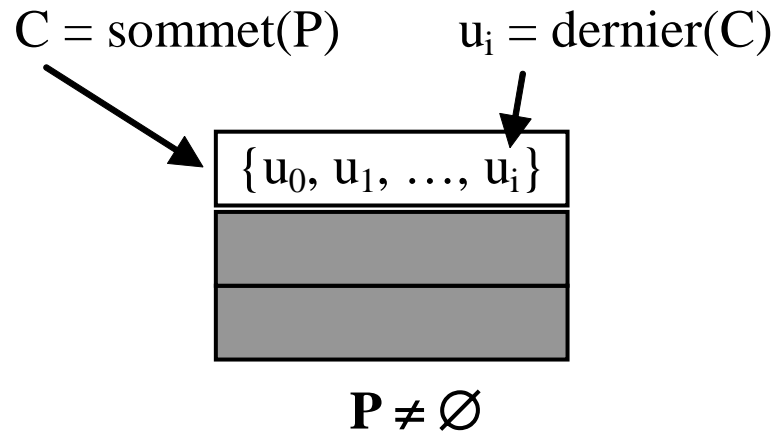
finsi

finpour

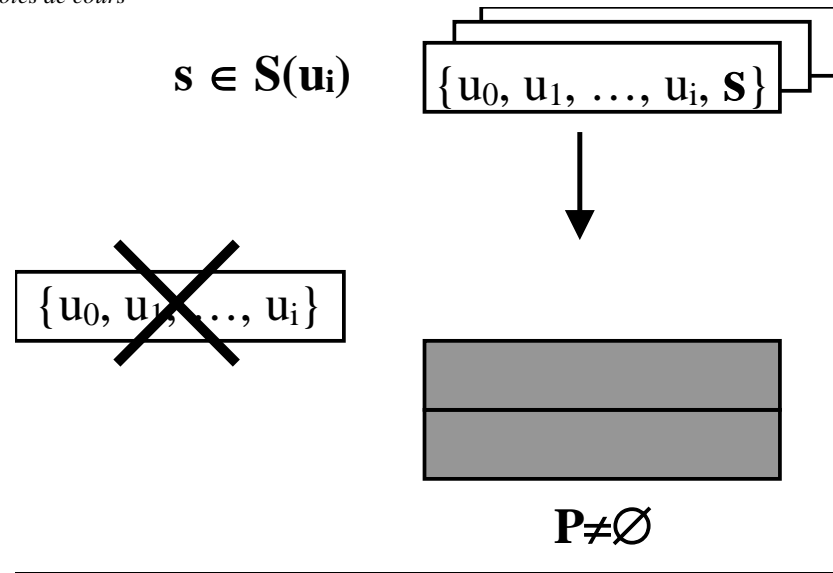
fintantque

3 si $P \neq \emptyset$ 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 (redondances)
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 allant de u_0 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 $\text{pere}(u_i)$ 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,...,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*(u₀,u_i) = coût du chemin optimal allant de u₀ à u (au départ il est inconnu)
- **g(u_i)** coût du **meilleur chemin connu** de u₀ à u_i. On aura g(u_i) = g*(u_i) lorsqu'à une itération donnée, le meilleur chemin entre u₀ et u_i aura été trouvé.
- **pere(u_i)** = pointeur indique l'état qui précède u_i sur le meilleur chemin connu depuis u₀.

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

```
1 P ← {u0} ; Q ← ∅ ; g(u0) ← 0 ; /* initialisations */
2 tantque P ≠ ∅ et sommet(P) ≠ T
    ui ← sommet(P) ; dépiler(P)
    Q ← Q ∪ {ui}
    déterminer S(ui)
    pour chaque s ∈ S(ui)
        si s ∉ P ∪ Q ou si g(s) > g(ui) + k(ui,s) /* soit s est un nouvel état soit on vient de trouver un meilleur chemin pour y arriver*/
            alors g(s) ← g(ui) + k(ui, s) /* actualisation de g(s) (diminution) */
                pere(s) ← ui /* mémorisation du meilleur prédécesseur de s
                insérer s dans P par ordre ↗ de g /* sinon s n'est pas mémorisé */
        finsi
    finpour
fintantque
3 si P ≠ ∅, alors ui = premier(P) et le chemin {u0 ,..., pere(pere(ui)), pere(ui), ui} 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 puisqu'ils ne sont pas au sommet de P).

Application sur l'exemple de la page 7

Initialisation

P	Q	Meilleurs coûts	Peres
S	\emptyset	$g(S)=0$	

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

P	Q	Meilleurs coûts	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 **totale**ment ordonné par $f \nearrow$

Les états ayant même valeur de f , sont ordonnés par $g \searrow$ (ou ce qui revient au même par $h \nearrow$).

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 $\text{père}(u)$ sur l'état qui le précède sur ce chemin.

On note : $\text{chemin}(u) = \{u_0, \dots, \text{père}(\text{père}(u)), \text{père}(u), u\}$

ALGORITHME A*

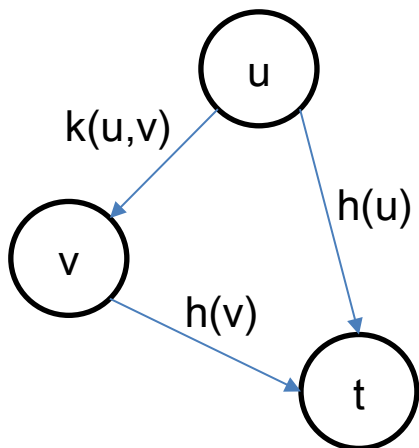
```
1 P ← {u0};    Q ← ∅;    g(u0) ← 0                /* Initialisations */
2 tantque P ≠ ∅ et sommet(P) ∉ T
    u ← premier(P) ; dépiler(P)
    Q ← Q ∪ {u}
    déterminer S(u)
    pour chaque s ∈ S(u)
        si s ∉ P ∪ Q                                /*s est nouveau (jamais atteint par l'algorithme)*/
        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) ← g(u) + k(u,s)                /*alors on a trouvé un chemin plus court pour aller en s : en passant par u */
            f(s) ← g(s) + h(s)
            pere(s) ← u                            /* 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 ≠ ∅,    alors t = premier(P) ∈ T et {u0 ,..., pere(pere(ui)), pere(ui), ui} 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 $\{u, v_1, v_2, \dots, v_q, t\}$ un chemin optimal de u vers un état terminal.

Comme h est monotone on peut écrire :

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

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

$$\dots \leq \dots$$

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

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

Comme h est coïncidente, $h(t) = 0$, et donc $h(u) \leq h^*(u) \Rightarrow h$ minorante 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 \Rightarrow 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 énumérés (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 réapparaît plus jamais 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 $C = (u_0, u_1, u_2, \dots, t)$ 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 $u_0, u_1, u_2, \dots, u_i$ 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 \hat{u} situé au sommet de P est tel que : $f(\hat{u}) \leq f^*(u_0)$.

Preuve

Par définition \hat{u} 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 :

$g(u_i) = g^*(u_i)$. \hat{u} 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 donc $f(\hat{u}) \leq f(u_i)$

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

Comme A^* s'arrête, il finit nécessairement par rencontrer un état \hat{u} terminal et $\text{chemin}(\hat{u})$ 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 coïncidente.

Complexité

Soit N le nombre d'états total. Dans le pire des cas, on trouve le bon chemin en dernier. Sa recherche est donc en $O(2^M)$, chaque état u étant développé autant de fois qu'il y a de chemins entre u_0 et u .

Si h est minorante, la complexité peut descendre à $O(N^2)$.

Le choix de l'état à développer est \hat{u} tel que $f(\hat{u}) \geq \max(f(v)) \quad v \in Q$;

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 $N = 4,3 \cdot 10^{19}$ 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 est particulier à un problème donné, c'est :

- **E** : la structure de données représentant un état, ex : $[[a, b, c,], [d, e, f], [g, h, _]]$
pour un taquin 3x3
- **cle(u)** : la fonction qui permet d'identifier de manière unique un état u, ex : « abcdefgh_ »
- **S(u)** : la fonction de transition qui détermine les états suivants d'un état u
- **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 un état u et un état terminal.

A chaque étape de l'algorithme 2 traitements distincts pèsent lourdement sur la complexité :

- **déterminer le meilleur état de P** à développer (= celui qui a la valeur $f(\hat{u})$ minimale)
 $O(1)$, si P est géré comme une **file de priorité (= un tas binaire)** ordonnée selon les valeurs de f puis de h (en cas d'égalité de f)
- **déterminer si un état v successeur de \hat{u} est déjà connu** : il faut rechercher v dans P et sinon dans Q, car il ne faut garder qu'un seul exemplaire de v (celui qui minimise f)

Il faut éviter des vecteurs (tableaux) sinon la recherche d'un élément coûte $O(n)$.

Il faut utiliser **une table de hachage** ($O(1)$), **un dictionnaire** ou un **arbre binaire de recherche équilibré (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.

B	H	B
A	F	D
G		E

Départ

A	B	C
H		D
G	F	E

Arrivée

II.3.4.1 LE PROBLEME

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

Objectif : minimiser le nombre d'actions à réaliser.

II.3.4.2 UNE MODELISATION

- chaque action revient à déplacer le "trou" (à **gauche**, à **droite**, vers le **haut**, vers le **bas**).
- $k(u,v) = 1 \quad \forall v \in \text{suc}(u)$: 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 : $h(t) = 0$ pour $\forall t \in T$: 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

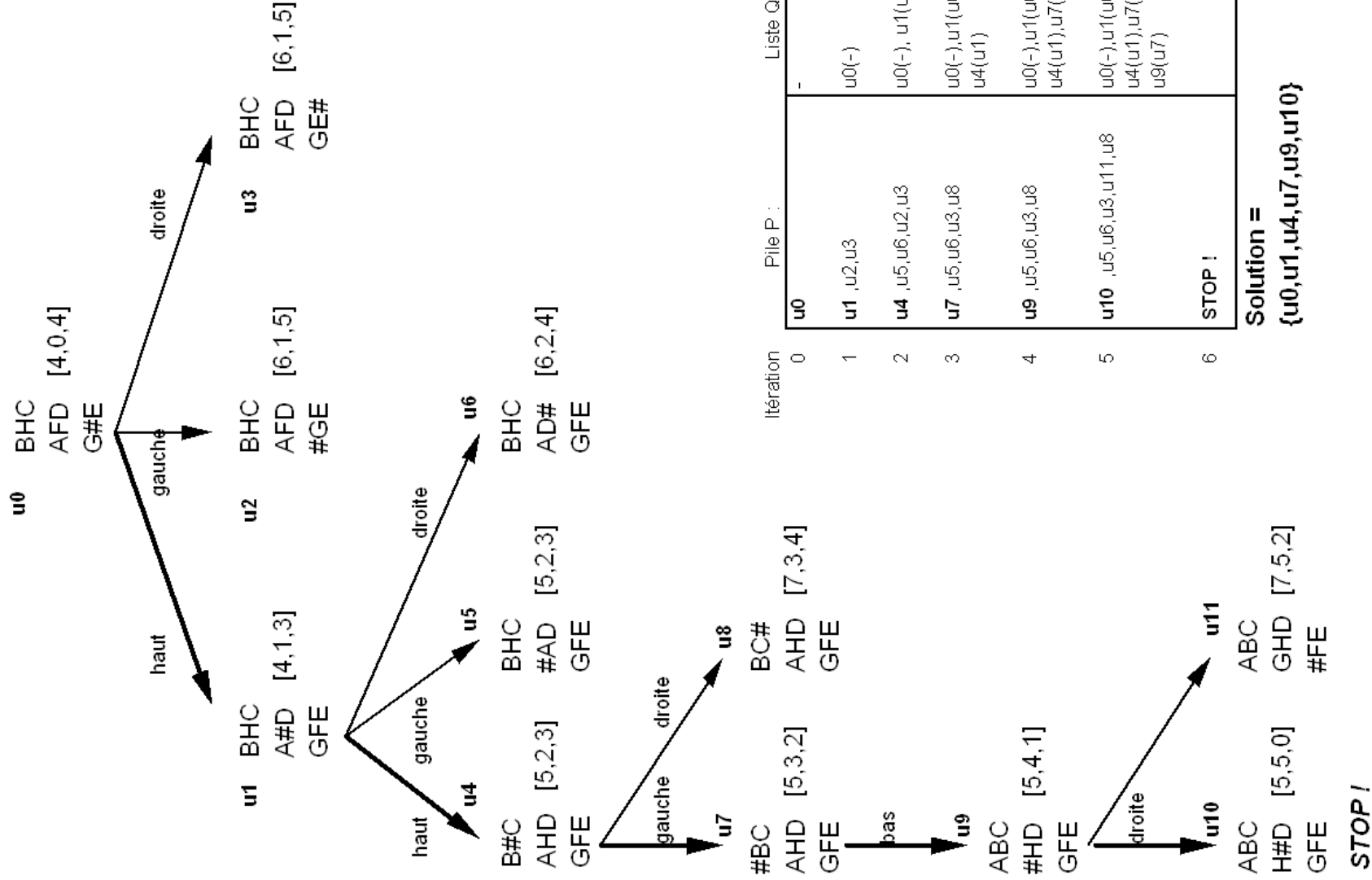
développement de u_4 , $S(u_4) = \{u_7, u_8\}$, etc

Cette numérotation permet de retrouver les étapes de A* à partir de l'arbre qu'il a généré et parcouru.

- 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 :

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

En TP on essaiera une deuxième heuristique bien meilleure, basée sur la *Distance de Manhattan*.



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

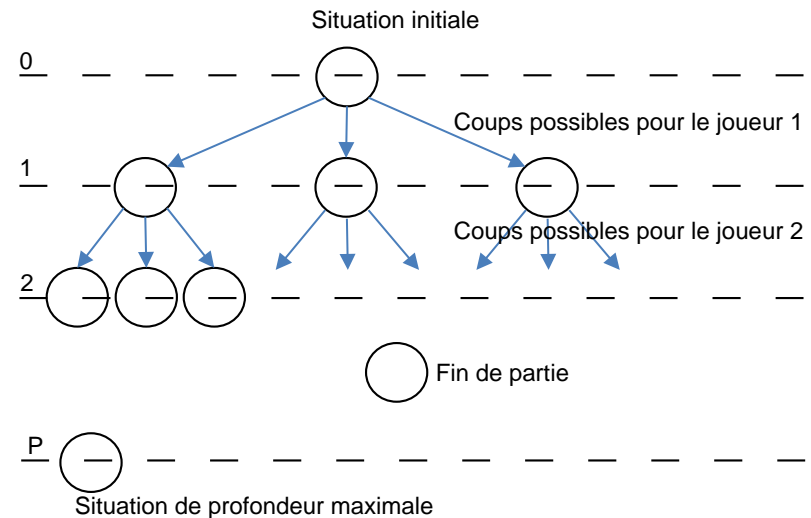
On considère des jeux à 2 joueurs :

- 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 prendre 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 $J' = \text{adversaire}(J)$

Convention Minmax :

On évalue toutes les situations **du seul point de vue du 1^{er} joueur** (celui qui joue à la racine). En une feuille, la fonction $h(u)$ mesure l'avantage apparent du **1^{er} joueur**. Exemple : aux Dames, les blancs commencent.

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

Le **1^{er} joueur est appelé max** (car il cherche à maximiser son gain) ; au contraire le 2^{ème} joueur est appelé min, car il cherche à minimiser le gain du 1^{er} joueur. Donc par convention :

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

Convention Negamax :

On évalue une situation u **du point de vue du joueur courant dans cette situation** (celui qui doit jouer en u)

La fonction $h(u, J)$ est la 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

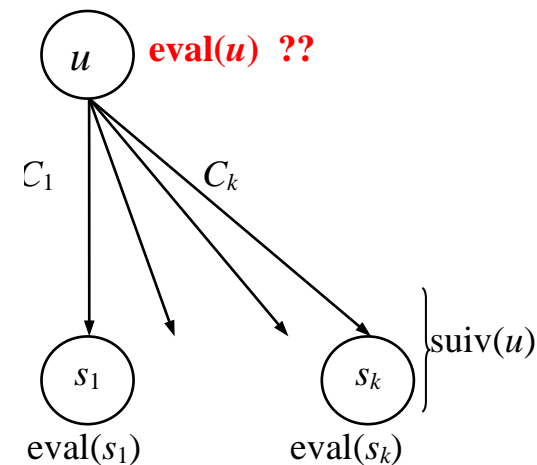
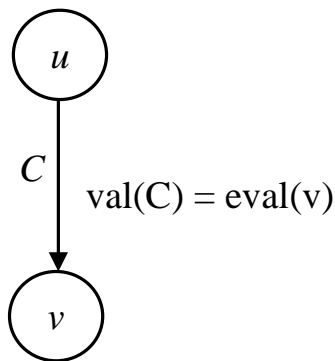
Pour une situation où Max a le tour on $h(u, \text{max}) = h(u)$; pour une où Min a le tour, $h(u, \text{min}) = -h(u)$

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 tous les coups jouables à partir de ce nœud et retenir le meilleur coup. La valeur d'un coup est la valeur de la situation atteinte par ce coup.



Si u = situation dans laquelle **max** (le 1^{er} joueur) doit jouer : $eval(u) = MAX_{s \in SUIV(u)}(eval(s))$

Si u = situation dans laquelle **min** (l'adversaire) doit jouer : $eval(u) = MIN_{s \in SUIV(u)}(eval(s))$

Si u est **terminale** (feuille), alors on utilise $h(u)$: $eval(u) = h(u)$

III.3.2 Cas particuliers

III.3.2.1 GAME OVER

On a : $\text{suiv}(u) = \emptyset$

$$\text{eval}(u) = h(u) = -\infty$$

perdant = max

gagnant = min

$$\text{eval}(u) = h(u) = +\infty$$

perdant = min

gagnant = max

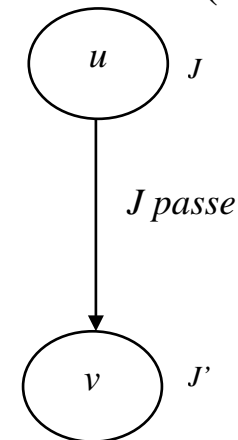
$$\text{eval}(u) = h(u) = 0$$

partie nulle

III.3.2.2 CAS DES JEUX DANS LESQUELS OU ON PEUT ETRE FORCE DE "PASSER SON TOUR" (EX : OTHELLO)

Si on accepte qu'un joueur « passe » son tour
dans ce cas, la situation u ne change pas : $v = u$

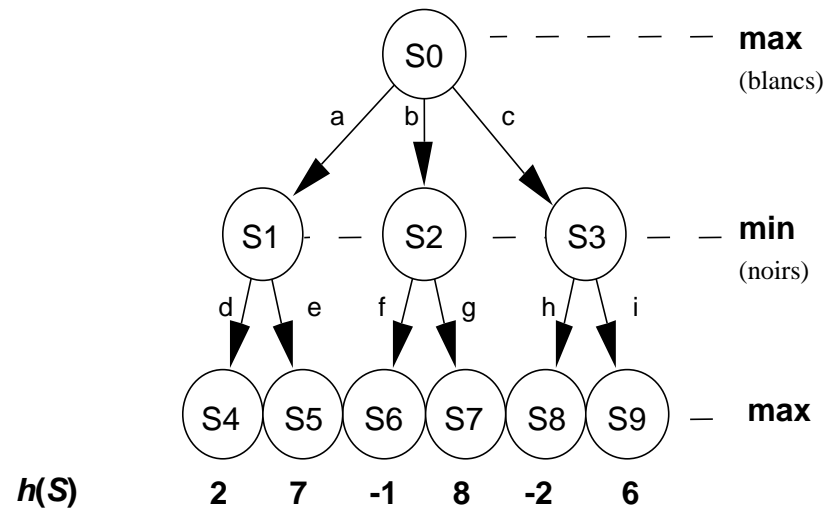
$$\text{eval}(u) = \text{eval}(v)$$



III.3.3 Application sur un exemple

ex : pour le jeu Othello, c'est toujours les blancs (=max) qui commencent ; on peut utiliser la fonction heuristique suivante :

$$h(S) = \text{nb de pions blancs} - \text{nb de pions noirs}$$



$$\text{eval}(S1) = \min(2,7) = 2 = \text{val_coup}(a)$$

$$\text{eval}(S2) = \min(1,8) = 1 = \text{val_coup}(b)$$

$$\text{eval}(S3) = \min(1,6) = 1 = \text{val_coup}(c)$$

$$\text{eval}(S0) = \max(2,1,1) = 2 = \text{val_coup}(a) \Rightarrow \text{a est le meilleur coup (pour une profondeur d'analyse de 2).}$$

III.3.4 L'algorithme minmax

III.3.4.1 CONVENTION NEGAMAX

Les définitions données en III.3.1 font implicitement intervenir le 1^{er} joueur (celui qui joue à la racine de l'arbre).

On évalue désormais une situation de façon **relative, du point de vue du joueur courant** (celui qui joue en u).

Pour une situation u développable :

$$eval(u, J) = MAX_{s \in SUIV(u)} (-eval(s, J'))$$

avec $J' = \text{adversaire}(J)$

Comme $\max(-f(x)) = -\min(f(x))$:

$$eval(u, J) = -MIN_{s \in SUIV(u)} (eval(s, J'))$$

Si u est une feuille ou une fin de partie :

$$eval(u) = h(u, J)$$

Dans la convention Negamax, on ne tient compte que du point de vue du joueur courant, donc :

$$h(u, J) = +\infty$$

si J gagne en u

$$h(u, J) = -\infty$$

si J perd en u

$$h(u, J) = 0$$

si partie nulle

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

$$h(u, J) = \text{nb de pions de } J - \text{nombre de pions de } J'$$

On a donc pour une situation u :

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

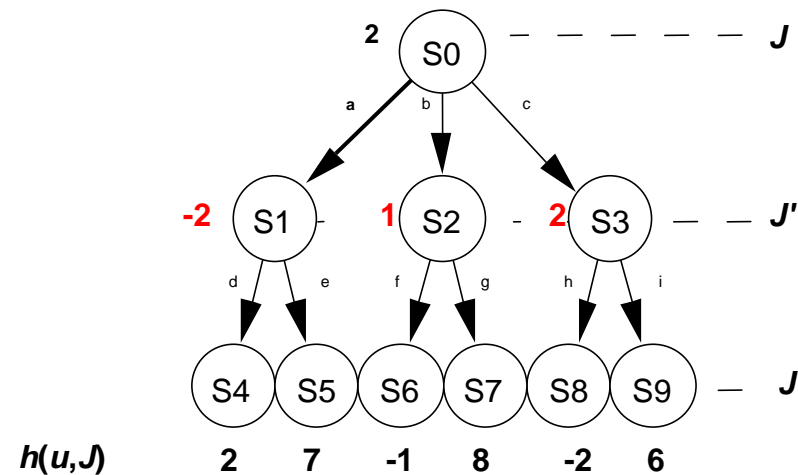
$$\underline{h}(u, \text{blancs}) = \text{nb de pions blancs en } u - \text{nb de pions noirs en } u = -h(u, \text{noirs})$$

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

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 la convention minmax.



III.3.4.3 ALGORITHME MINMAX

Dans la procédure qui suit :

- J joueur qui joue en u .
- P profondeur (nb de coups) sur laquelle on peut encore développer l'analyse
- C^* meilleur coup que doit jouer J en u
- F^* évaluation de u = évaluation de C^*

On note :

- $\downarrow A$ A argument donné à l'appel (de type *in*)
- $\uparrow A$ A argument retourné (de type *out*)

```

fonction MINMAX( $\downarrow J, \downarrow u, \downarrow P$ )  retourne une structure de type (Coup, Valeur)
   $C^* \leftarrow none$ 
  si  $P = 0$  ou si fin_de_partie( $u$ ) alors
     $F^* \leftarrow h(u, J)$  // pas de coup jouable on utilise alors la fonction heuristique pour évaluer la situation
  sinon
    si  $J = \text{MAX}$  alors
       $F^* \leftarrow -\infty$ 
      pour  $Coup \in \text{Coups\_Jouables}(J, u)$ 
         $s \leftarrow \text{Jouer}(J, Coup, u)$ 
         $(Adv\_coup, Adv\_score) \leftarrow \text{MINMAX}(\text{MIN}, s, P-1)$ 
        si  $Adv\_score > F^*$  alors
           $C^* \leftarrow Coup$  et  $F^* \leftarrow Adv\_score$  // mise à jour du coup qui amène au maximum
        finsi
      finpour
    sinon
       $F^* \leftarrow +\infty$ 
      pour  $Coup \in \text{Coups\_Jouables}(J, u)$ 
         $s \leftarrow \text{Jouer}(J, Coup, u)$ 
         $(Adv\_coup, Adv\_score) \leftarrow \text{MINMAX}(\text{MAX}, s, P-1)$ 
        si  $Adv\_score < F^*$  alors
           $C^* \leftarrow Coup$  et  $F^* \leftarrow Adv\_score$  // mise à jour du coup qui amène au maximum
        finsi
      finpour
    finsi
  retourner ( $C^*, F^*$ )
fin

```

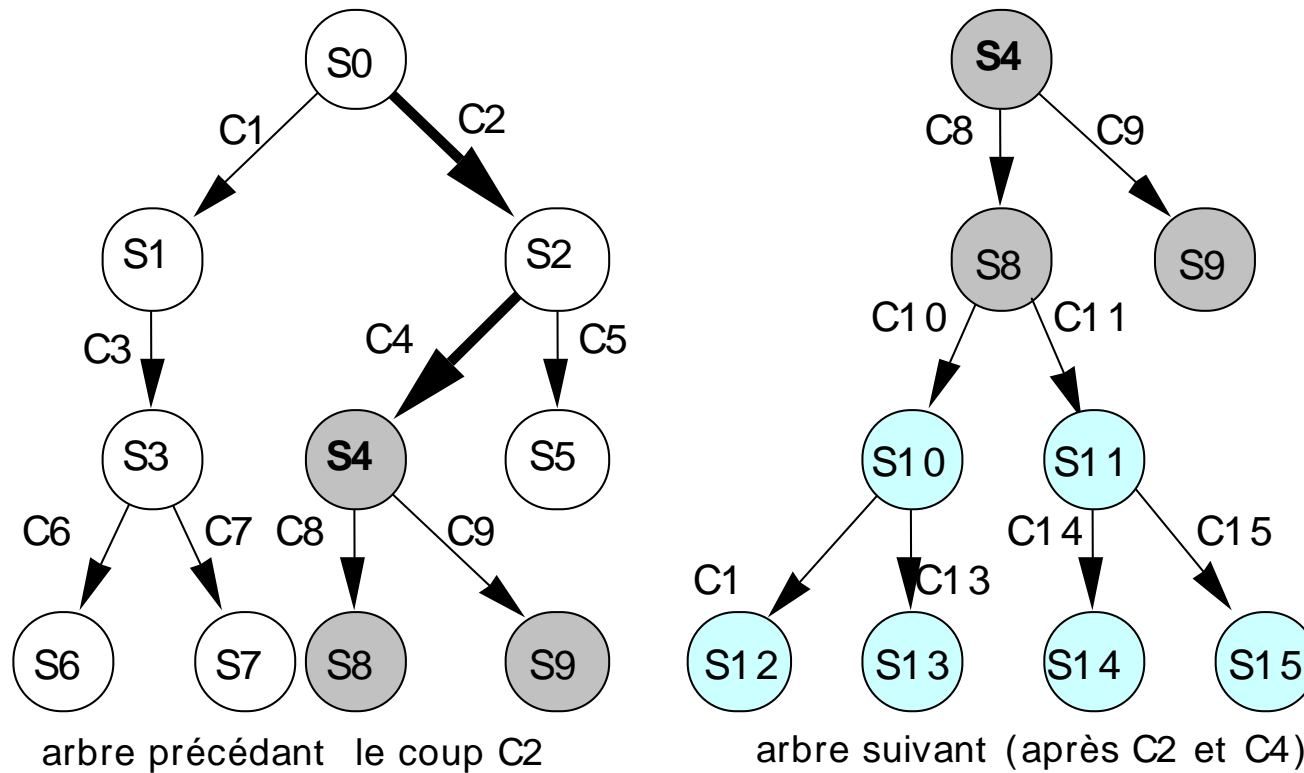
III.3.4.4 ALGORITHME MINMAX EN CONVENTION NEGAMAX

```
fonction Negamax( $\downarrow J, \downarrow u, \downarrow P$ )  retourne une structure de type (Coup, Valeur)
   $C^* \leftarrow none$ 
  si  $P = 0$  ou si fin_de_partie( $u$ ) alors
     $F^* \leftarrow h(u, J)$            // pas de coup jouable si profondeur maximale atteinte ou fin de partie
    // on utilise alors la fonction heuristique pour évaluer la situation
  sinon
     $F^* \leftarrow -\infty$ 
    pour  $Coup \in Coups\_Jouables(J, u)$ 
       $s \leftarrow Jouer(J, Coup, u)$ 
       $(Adv\_coup, Adv\_score) \leftarrow Negamax(J, s, P-1)$ 
      si  $-Adv\_score > F^*$  alors
         $C^* \leftarrow Coup$            // mise à jour du coup qui amène au maximum
      finsi
    finpour
  finsi
  retourner( $C^*, F^*$ )
fin
```

III.3.5 Algorithme pour joueur automatique

A l'aide de l'algo Minimax, 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).



Procédure 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(Cutilisateur) // on lit le coup
    u ← jouer(Cutilisateur, 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 la riposte de l'utilisateur
      lire(Cutilisateur)
      u ← jouer(Cutilisateur, 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 (x ou o sur une même ligne, colonne, ou diagonale), dans une grille carrée N×N, initialement vide.

On se limite au cas N=3.

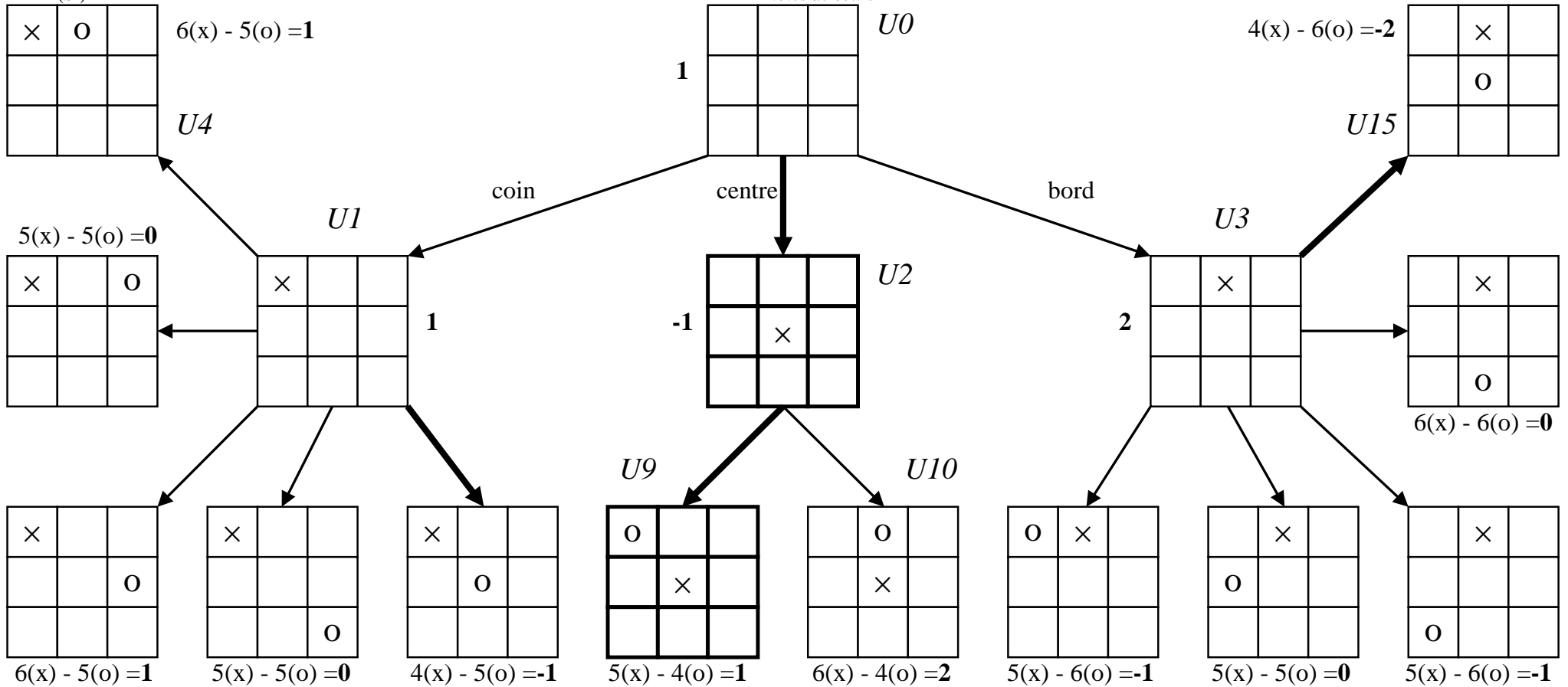
On tire au sort le premier joueur : (x) commence.

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

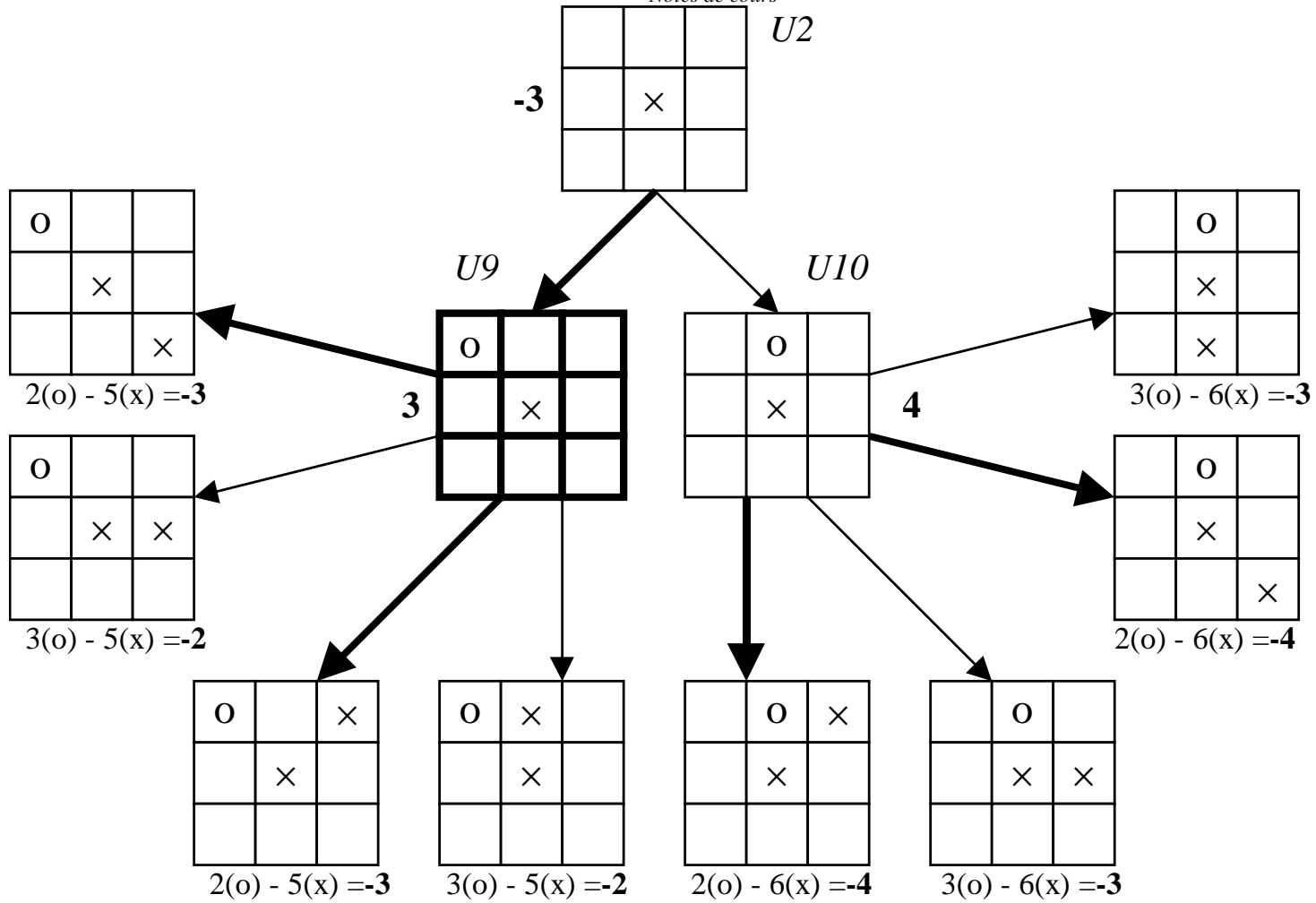
Une heuristique possible :

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) :

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



Evaluation negamax du meilleur coup initial pour les 'x'



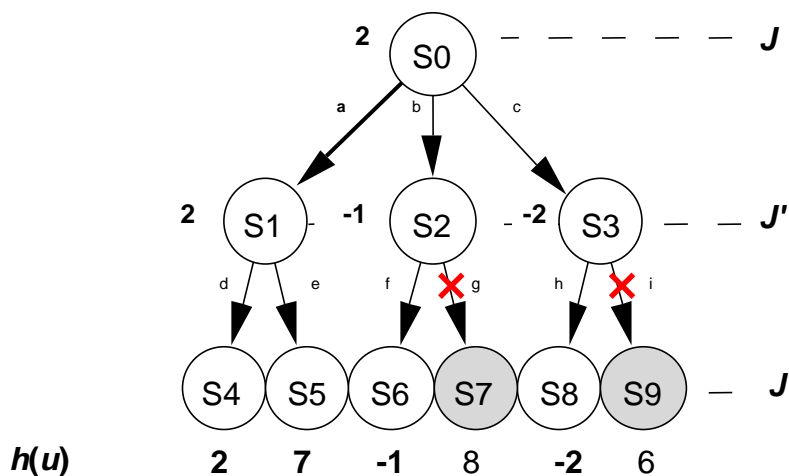
Evaluation de la meilleure riposte pour les 'o'

III.4 L'élagage alpha-beta

III.4.1 Critique de l'algorithme 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	$S0 \rightarrow$	$S1 \rightarrow$	$S4 \rightarrow$	$S1 \rightarrow$	$S5 \rightarrow$	$S1 \rightarrow$	$S0$
$val(u)$?	?	$=2$	≤ 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 $\text{val}(S2) \leq 2$. Idem pour S3.

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

situation	S0 →	S2 →	S6 →	S2 →	S7
val(<i>u</i>)	≥ 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	S0 →	S3 →	S8 →	S3 →	S9
val(<i>u</i>)	≥ 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 de l'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 $\alpha = -\infty$

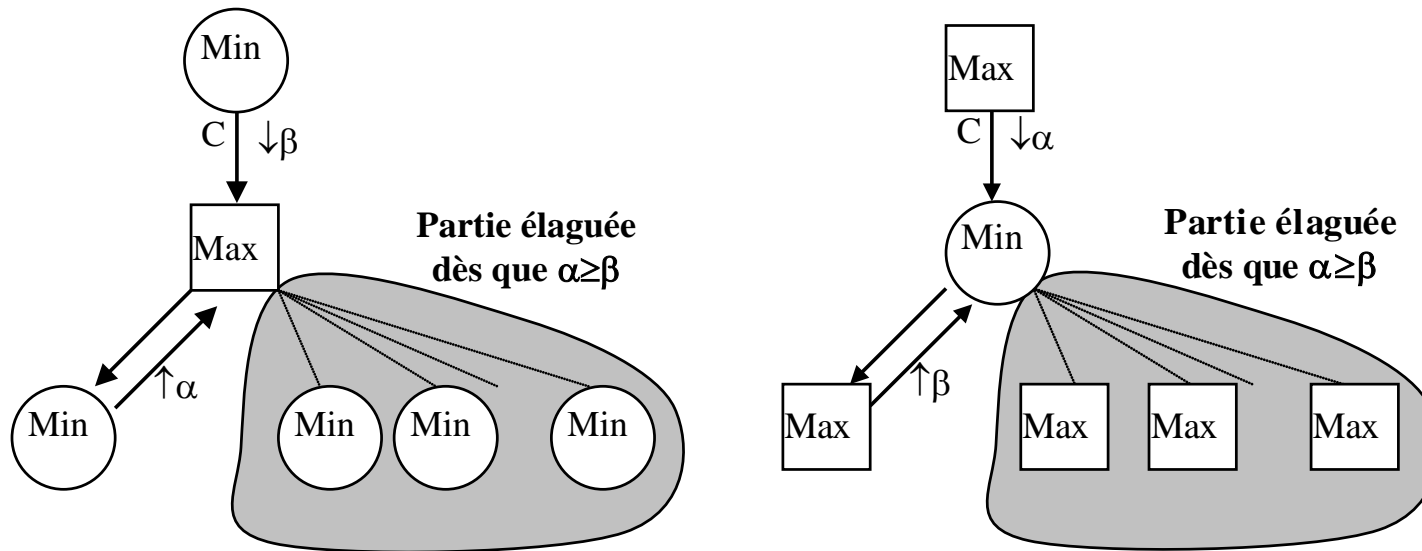
On appelle β -valeur d'un nœud de type Min, une **borne maximale courante** de la valeur de ce nœud, après exploration d'un certain nombre des nœuds suivants. Si aucun suivant n'a encore été exploré $\beta = +\infty$

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 remonte d'un de ses fils dépasse la valeur β (resp. α) qu'il a reçue de son père.



III.4.3 AlphaBeta

On présente d'abord la version sans convention negamax, puis avec

III.4.3.1 VERSION SANS CONVENTION NEGAMAX


```

function ALPHABETA(Player, Situation, Depth, A, B) // without NEGAMAX convention
  Alpha = A
  Beta = B
  if Depth == 0 or else Game_Over(Player, Situation) then // here
    Best_Move = NONE // no move is enabled
    Score = Static_Heuristic_Evaluation(Player, Situation) // the static evaluation is called
  else
    Possible_Moves = Evaluate_Moves(Player, Situation) // sorted by decreasing interest
    Prune = FALSE // if TRUE, no further situations will be evaluated
    if Player == MAX then
      Best_Move = NONE
      Score = MINIMUM_VALUE // a very negative value, say -10000000
      while Possible_Moves /= EMPTY and not Prune loop
        Move = First(Possible_Moves)
        Possible_Moves = Tail(Possible_Moves)
        Next_Sit = Play(Player, Move, Situation)
        (Adv_Move, Adv_Score) = ALPHABETA(MIN, Next_Sit, Depth-1, Alpha, Beta)
        // returns a 2-fields structure

        if Adv_Score > Score then
          Best_Move = Move
        endif
        Score = max(Score, Adv_Score)
        Alpha = max(Alpha, Adv_Score)
        Prune = (Alpha >= Beta)
      endloop

```

```
else // Player = MIN
  Best_Move = NONE
  Score = MAXIMUM_VALUE // a very positive value , say +10000000
  while Possible_Moves /= EMPTY and not Prune loop
    Move = First(Possible_Moves)
    Possible_Moves = Tail(Possible_Moves)
    Next_Sit = Play(Player, Move, Situation)
    (Adv_Move, Adv_Score) = ALPHABETA(MAX, Next_Sit, Depth-1, Alpha, Beta)
    // returns a 2-fields structure

    if Adv_Score < Score then
      Best_Move = Move
    endif
    Score = min(Score, Adv_Score)
    Alpha = min(Alpha, Adv_Score)
    Prune = (Alpha >= Beta)
  endloop
endif
endif
return (Best_Move, Score) // returns a 2-fields structure
end ALPHABETA
```

III.4.3.2 VERSION AVEC CONVENTION NEGAMAX

```

function NEGAMAX_ALPHABETA(Player, Situation, Depth, A, B)           // With NEGAMAX convention
    Alpha = A
    Beta  = B
    Opp   = Opposite(Player)
    if Depth == 0 or else Game_Over(Player, Situation) then           // here
        Best_Move = NONE                                             // no move is enabled
        Score      = Static_Heuristic_Evaluation(Player, Situation) // the static evaluation is called
    else
        Possible_Moves = Evaluate_Moves(Player, Situation)           // sorted by decreasing interest
        Prune          = FALSE                                       // if TRUE, no more next situations will be evaluated
        Best_Move = NONE
        Score      = MINIMUM_VALUE                                  // a very negative value, say -10000000
        while Possible_Moves /= EMPTY and not Prune loop
            Move          = First(Possible_Moves)
            Possible_Moves = Tail(Possible_Moves)
            Next_Sit      = Play(Player, Move, Situation)
            (Adv_Move, Adv_Score) = NEGAMAX_ALPHABETA(Opp, Next_Sit, Depth-1, -Beta, -Alpha)
            if -Adv_Score > Score then
                Best_Move = Move
            endif
            Score = max(Score, -Adv_Score)
            Alpha = max(Alpha, -Adv_Score)
            Prune = (Alpha >= Beta)
        endloop
    endif
    return (Best_Move, Score)
end NEGAMAX_ALPHABETA

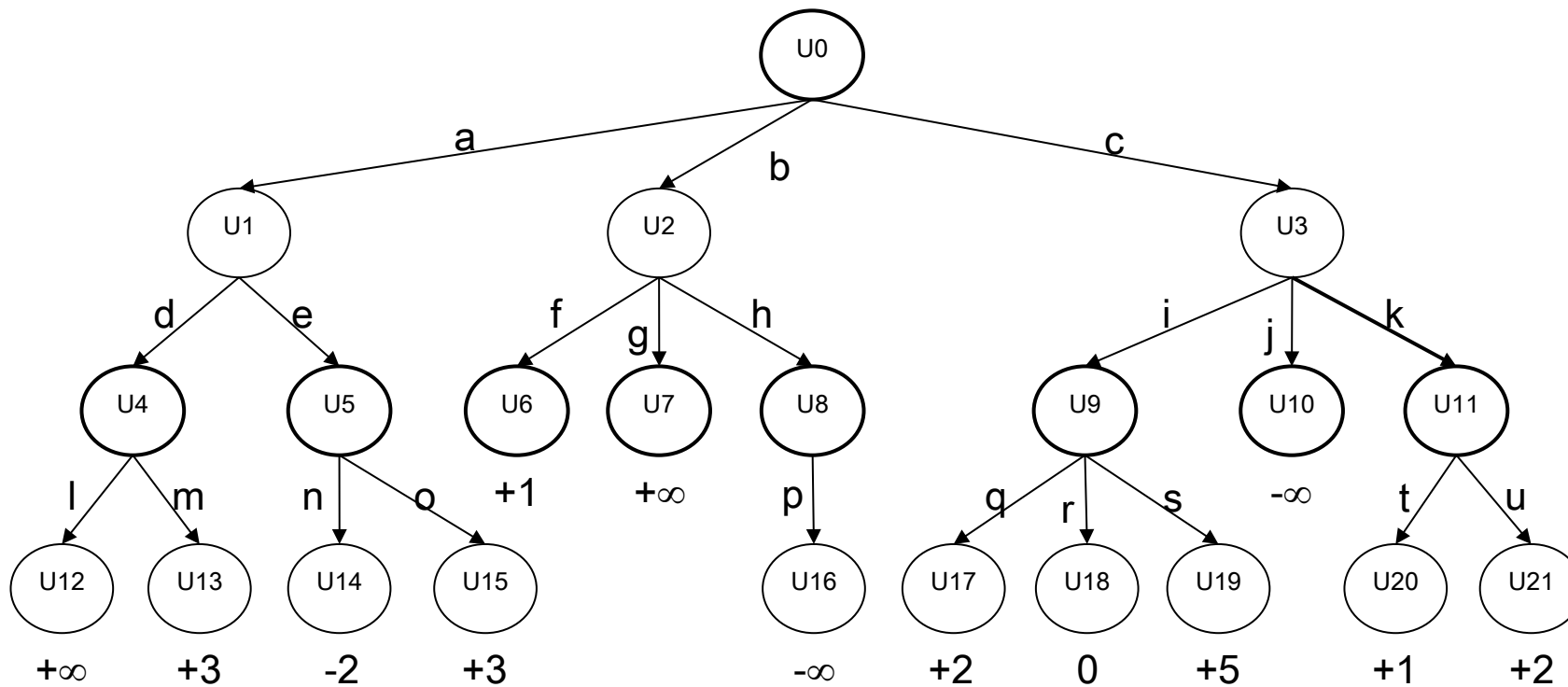
```

III.4.4 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 $-\infty$ et $+\infty$ dans certains nœuds (qui perd ou gagne ?)
- b. établir la valeur minmax de la situation initiale U_0 et le meilleur coup à jouer pour le 1^{er} 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 U_0 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.

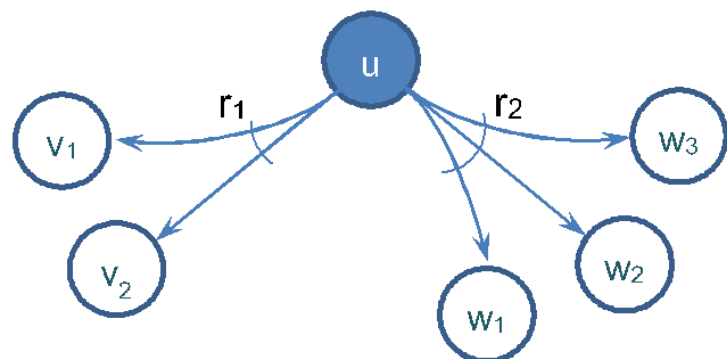
IV Recherche heuristique dans les graphes de sous-problèmes

IV.1 Formulation générale

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

- la **racine** (nœud sans prédécesseur) représente le problème à décomposer
- une **règle de décomposition** relie 1 sommet père(=problème) à k sommets fils (=sous-problèmes) ; c'est un **k -connecteur** (1 origine, k extrémités) ou **hyper-arc**. Un problème donné peut être décomposé de plusieurs façons (plusieurs règles sont applicables). Le graphe résultant est un **hyper-graphe**.

Exemple



2 façons de décomposer $u \Rightarrow$ 2 hyper-arcs.

r_1 est un 2-connecteur.

r_2 est un 3-connecteur .

- les **feuilles** représentent des **problèmes** triviaux **résolus** (qui n'ont pas à être décomposés).
- la **solution** n'est pas un chemin, c'est un **sous-graphe**.

IV.2 Hyper-graphes et graphes ET/OU

Un hyper-graphe peut également être représenté sous forme de graphe ET-OU.

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 : il est résolu si tous les sous-problèmes sont résolus.
- un **nœud OU** représente un problème décomposable en une **disjonction** de sous-problèmes ; il est résolu si au moins 1 des sous-problèmes est résolu

IV.2.1 Exemple

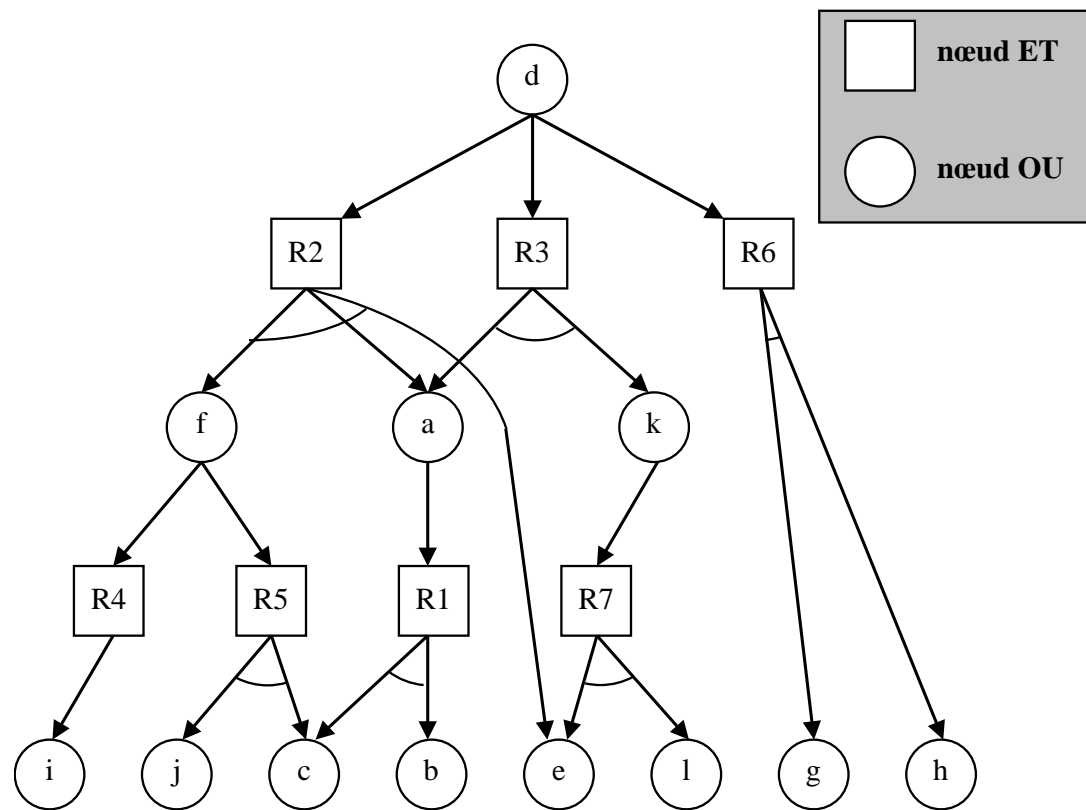
Soit le système de production :

R1 : a :- b, c. R5 : f :- c, j.

R2 : d :- a, e, f. R6 : d :- g, h.

R3 : d :- a, k. R7 : k :- e, l.

R4 : f :- i.

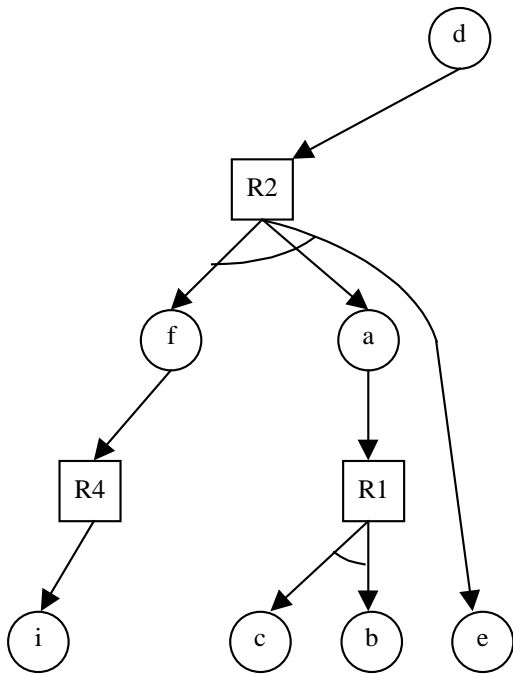


Un graphe ET/OU

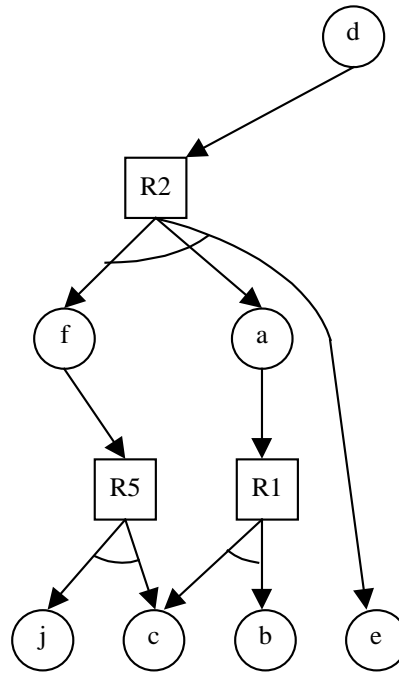
IV.2.2 Sous-graphes solutions

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

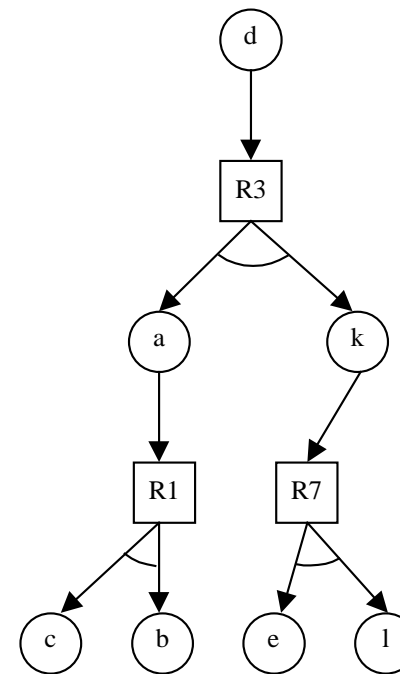
- la racine du graphe initial
- si un nœud ET appartient à cette solution, tous ses nœuds fils appartiennent aussi à la solution,
- si un nœud OU appartient à cette solution, un de ses nœuds fils appartient aussi à la solution.



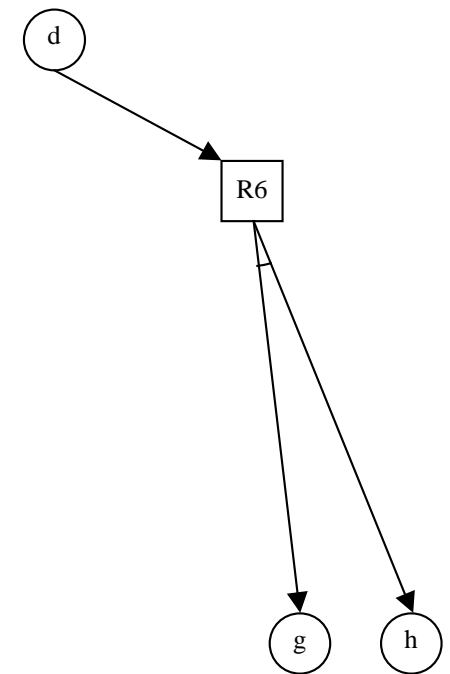
solution n°1



solution n°2



solution n°3



solution n°4

Quel est la solution **optimale** ? Cela dépend de l'interprétation donnée au graphe de décomposition.

Si c'est le graphe de résolution possible d'une proposition logique en Prolog, on a des clauses de 2 types :

des règles, du type : `pb :- sp1, sp2, ..., spN.`

des faits, du type : `fact.`

On associe Un nœud ET par règle et un nœud OU par proposition

On cherche la démonstration optimale qui minimise le nombre de faits nécessaires pour résoudre un but donné, ex : résoudre d.

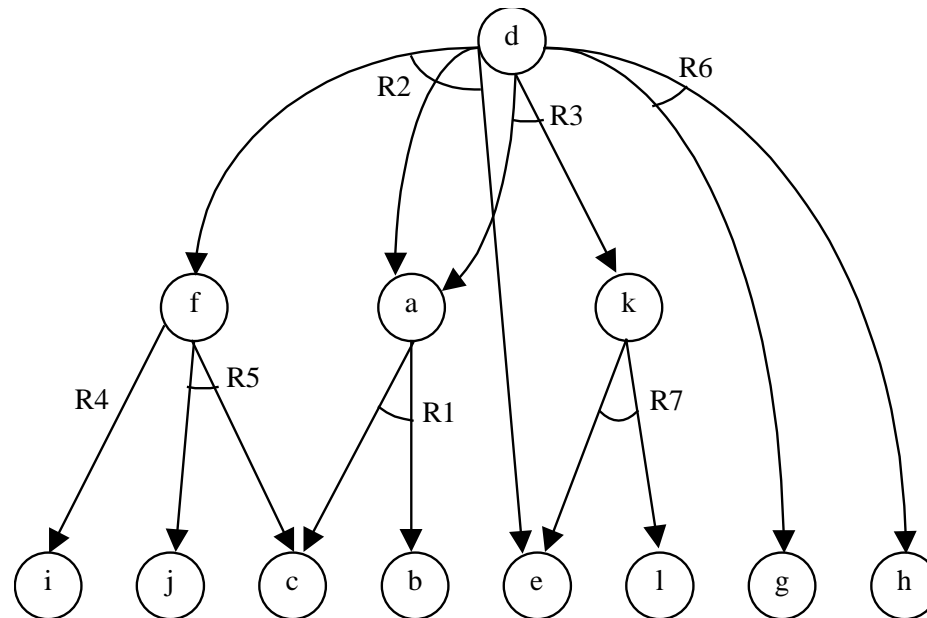
Il existe 4 bases de faits, une par solution : $\{j, c, b, e\}$, $\{j, c, b, e\}$, $\{c, b, e, l\}$, $\{g, h\}$.

La solution optimale est la solution n° 4, qui ne fait intervenir que 2 faits : $\{g, h\}$

Plus généralement, on s'intéresse aux sous-graphes de coût total minimal et le coût est lié essentiellement à chaque règle. Comment définir le coût d'une solution ?

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)
- $R(u)$: règles (ou k -connecteurs) permettant de décomposer u
- $S(u, r)$: successeurs de u par la règle $r \in R(u)$
- $s_{r,i}$: $i^{\text{ème}}$ successeur après application de r
- $c(r)$: coût de la règle r
- $p(u, r)$: prédécesseur de u par la règle r

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

On suppose que l'hyper-graphe est acyclique : tout nœud v 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 T (problèmes résolus) est défini récursivement :

si u est résolu ($u \in T$) alors

$$k(u) = 0$$

sinon

$$k(u) = \min (c(r) + k(s_1) + k(s_2) + \dots + k(s_i) + \dots + k(s_{nr}))$$

$$r \in R(u), s_i \in S(u, r)$$

k est la fonction exacte, mais n'est connue qu'en fin d'algorithme. A une étape intermédiaire donnée, on estime k par la fonction q .

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 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 donc on peut arrêter l'exploration.

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

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** :

$$h(n) = 0 \text{ pour tout nœud terminal}$$

et **monotone** :

$$h(u) \leq c(r) + h(s_1) + h(s_2) + \dots + h(s_m) \quad \forall r \in R(u) \text{ et } S(u,r) = \{s_1, s_2, \dots, s_m\}$$

est **minorante**.

Principes de l'algorithme AO*

Choix du nœud à développer : un nœud non encore développé qui est extrémité d'un connecteur marqué (ou la racine au tout début). Si plusieurs candidats, prendre un nœud tel que $h(u)$ est minimale.

- avant expansion d'un nœud :

$$q(u) = h(u)$$

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

$$q(u) = \min_{r \in R(u)} \left(c(r) + q(s_{r,1}) + \dots + q(s_{r,n}) \right)$$

Procedure AO*

// Initialisation

- $G \leftarrow \{n_0\}$ (problème à résoudre).
- $q(n_0) \leftarrow h(n_0)$
- si n_0 est un problème terminal, marquer n_0 résolu

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

tantque n_0 n'est pas marqué résolu

// expansion d'un nœud (exploration descendante)

calculer G' , sous-graphe solution partiel en suivant tous les connecteurs **marqués** de G depuis n_0

sélectionner un nœud non développé (feuille) **n** de G'

déterminer $R(n)$ et tous les successeurs de **n**

pour chaque successeur s_j de **n**

si s_j n'est pas déjà dans G **alors**

$G \leftarrow G \cup \{s_j\}$

// ici le graphe G se développe

$q(s_j) \leftarrow h(s_j)$

// évaluation par défaut

si s_j est un problème terminal alors $\text{resolu}(s_j) \leftarrow \text{VRAI}$

fsi

fsi

fp

// révision des coûts (exploration ascendante)

$S \leftarrow \{n\}$

tantque $S \neq \emptyset$

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 $\{s_{r1}, s_{r2}, \dots, s_{rk}\}$

calculer $q(m) = \min_{r \in R(u)} (c(u, r) + q(n_{r1}) + \dots + q(n_{rk}))$

marquer le connecteur r qui permet d'obtenir ce minimum

démarquer éventuellement le connecteur précédemment marqué issu de m

si tous les successeurs $\{n_{r1}, n_{r2}, \dots, n_{rk}\}$ sont marqués résolus **alors**

$resolu(m) \leftarrow \text{VRAI}$

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

