Exploration systématique de graphes

A. Introduction

Les techniques d'exploration de graphes sont des algorithmes de résolution de problèmes à usage général, très utilisés dans le domaine de l'intelligence artificielle.

Dans ce type de problèmes (par exemple : trouver la sortie dans un labyrinthe), la recherche de la solution se résume en un parcours de graphe (appelé Espace de recherche) pour trouver un chemin, s'il existe, entre le sommet initial (l'état ou la configuration initiale du problème) et un sommet solution représentant une configuration où le problème est résolu (l'état finale du problème).

Le passage d'un état à un autre (représenté par un arc entre deux sommets) est réalisé par application de règles définissant les propriétés et caractéristiques du problème à résoudre.

Les recherches en profondeur et en largeur sont des méthodes d'exploration (fouille) systématique pour la recherche d'une solution dans un graphe orienté implicite, le plus souvent sans circuit ou même arborescent.

La recherche en profondeur consomme généralement beaucoup moins d'espace que la recherche en largeur.

Par contre si le graphe est très grand (ou infini) ou si l'on veut plutôt chercher la solution la plus proche du sommet initial (en nombre d'arcs traversés), la recherche en largeur est alors nécessaire.

B. Recherche en profondeur (Backtracking)

La technique consiste à se déplacer dans l'espace de recherche en suivant un parcours en profondeur jusqu'à atteindre un état solution. Si la solution n'existe pas et si l'espace de recherche est fini, l'algorithme s'arrêtera après avoir explorer tous les états.

1. Exemples

a. Problèmes des *8 reines* :

"Il faut placer 8 reines dans un échiquier (matrice 8X8) sans qu'aucune d'entres elles ne soit en prise par une autre." Deux reines sont en prise, si elles se trouvent sur une même ligne, une même colonne ou une même diagonale).

Donc une reine placer dans la case d'indice (i,j) condamnera la ligne i, la colonne j et les deux

diagonales passant par la case (i,j).

La diagonale positive (DiagP) est celle qui forme un angle de +45   et la diagonale négative (DiagN) est celle qui forme un angle de -45   .

Les états formant l'espace de recherche sont les différents échiquiers avec un nombre de reines déjà placées variant entre 0 et 8. Donc l'espace est très grand mais fini.

L'état initial représente un échiquier vide. Les états solutions sont ceux représentant des échiquiers avec 8 reines déjà placées sans qu'il y ait de prise entre elles. A chaque transition d'un état à un autre on doit placer une nouvelle reine dans l'échiquier sans provoquer de conflit avec les reines déjà placées.

Voici le déroulement partiel d'une recherche en profondeur pour ce problème:
On démarre d'un échiquier vide.
premier niveau : Il ya 64 (ou 8*8) cas possibles de placer la première reine.
Deuxième niveau : Pour chaque cas du niveau 1, il ya moins de 63 possibilités de placer une reine. (Chaque cas qui ne répond pas à la condition est écarté).
Troisième niveau : on essaye de placer la troisième reine sans qu'il y ait de prise avec les deux premières.
Et ainsi de suite....

Si à un moment donné on ne peut plus placer une nouvelle reine sans provoquer de conflit, on fait un retour arrière pour remonter au dernier niveau où il y avait une autre alternative et on continue à partir de cette nouvelle branche.

On peut améliorer l'algorithme en remarquant que chaque ligne d'un état solution, contient exactement une seule reine. Donc à chaque ligne on doit place une reine, cela réduit considérablement l'espace de recherche et donc le temps d'exécution.

On peut ainsi représenter tout l'échiquier par un vecteur V[1, 8] où les indices désignent les lignes et les contenus les colonnes des reines déjà placées.

Par exemple si V est le vecteur {1, 3, 7,....}, cela voudrais dire que :
Première ligne, première colonne: on a une reine
Deuxième ligne, troisième colonne : on a une reine
Troisième ligne, septième colonne : on a une reine
...
Afin de montrer le principe, restreignons l'espace de recherche en essayant de placer 4 reines dans un échiquier 4 X 4.

Un arbre implicite se construit avec une recherche en profondeur.
Niveau 0 : l'echiquier vide est représenté par le vecteur vide V.
Niveau 1 : placer la reine sur la première colonne de la ligne une (V[1]=1).

Niveau 2 : essayer de placer la seconde reine de sorte qu'il n'y ait pas de prise. La troisième colonne convient (V[1]=1 et V[2]=3)
Niveau 3 : Echec, on ne peut pas placer la troisième reine dans la troisième ligne. Aucune colonne ne convient.
Donc on remonte au niveau 2 et on essaye de placer la deuxième reine ailleurs.
Niveau 2 : V[1] = 1 et V[2] = 4.
Niveau 3 : On peut maintenant placer la troisième reine à la colonne 2: V[1]=1, V[2]=4, V[3]=2
Et ainsi de suite ....

A chaque échec, on remonte dans l'arbre pour essayer d'autres éventualités.

Remarquer que :
- pour une diagonale négative, la différence des indices est constante.
- pour une diagonale positive, la somme des indices constante

Donc à chaque fois qu'on place une reine sur une case (i,j), on condamne la ligne i, la colonne j, l'ensemble des cases dont la somme des indices est égale à (i+j) (la diagonale positive) et l'ensemble des cases dont la différence des indices est égale à (i-j) (la diagonale négative).

La procedure qui suit place les 8 reines dans l'échiquier :

```
Procedure Placer (K , Col, DiagP, DiagN)
{On a déjà réussi à placer K reines et on place les reines suivantes,}
Si K = 8 :
        Ecrire (V)
Sinon
        Pour j = 1,8 :
                Si (j    Col ET  k+1 + j    DiagP ET k+1 - j    DiagN) :
        /* on place une nouvelle reine à la ligne K+1 et la colonne j */
        V[k+1] := j
                                Col := Col U {j}
                                DiagP := Diagp U {k+1 + j}
                                DiagN := DiagN U {k+1 - j}
                                Placer( k+1, Col, DiagP, DiagN )
                Fsi
        Finpour
Fsi
/* Col, DiagP et DiagN sont les ensembles de cases déjà condamnées.*/
```

Avec Placer(0, {}, {}, {}) comme appel initial.
Col, DiagP et DiagN sont initialement vides {}.

Remarquer que cet algorithme ressemble à l'algorithme DFS défini sur les parcours de graphes.

b- Problèmes des *cruches d'eau.*

Soient deux cruches A et B avec des capacités respectives de 4 et 3 litres.
A l'état initial les deux cruches sont vides, et par une série de manipulations on veut obtenir 2 litres d'eau dans la cruche A.

Les manipulations permises sont données par les règles suivantes:

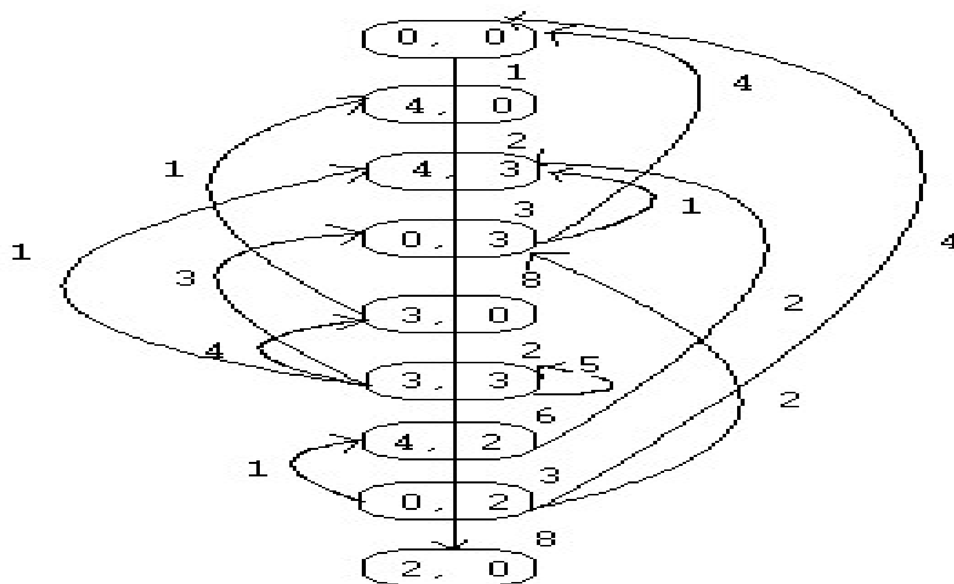
1. on peut remplir A Si A est non pleine
2. on peut remplir B Si B est non pleine
3. on peut vider A Si A est non vide
4. on peut vider B Si B est non vide
5. on peut verser le contenu de A dans B jusqu'à ce que B soit pleine Si A est non vide et Q(A) > (3-Q(B)).
6. on peut verser le contenu de B dans A jusqu'à ce que A soit pleine Si B est non vide et Q(B) > (4-Q(A)).
7. on peut verser tout le contenu de A dans B Si A est non vide et Q(A)   (3-Q(B)).
8. on peut verser tout le contenu de B dans A Si B est non vide et Q(B)   (4-Q(A)).

Q(c) représente la quantité d'eau courante dans la cruche c.

Un état de l'espace de recherche peut être représenté par un couple (x,y) où x est la quantité d'eau contenu dans la cruche A et y la quantité d'eau dans la cruche B.
L'état initial est donc : (0,0).
Les états solution ont la forme : (2, n) avec n quelconque.

La technique du backtracking consiste à appliquer les règles de 1 à 8, (dans cet ordre par exemple), pour chaque nouvel état visité.

Voici une partie de l'espace de recherche exploré par la technique du backtracking depuis l'état initial (0,0) jusqu'à un état solution (2,0):

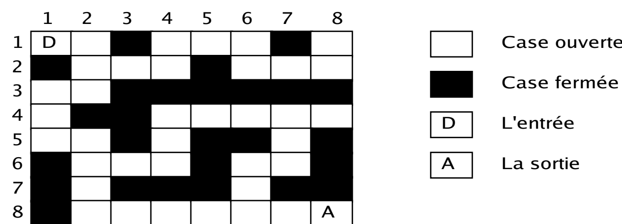


Pour éviter de passer plusieurs fois par un même état, il faudra garder la trace de tous les états déjà visités. Dans le graphe ci dessus, les arcs montant mènent vers des états déjà visités. A chaque fois qu'on génère un état déjà visité, on abandonne la branche courante pour essayer une autre alternative (retour arrière).

c- Trouver la sortie dans un labyrinthe :

Le problème consiste à trouver un chemin, s'il existe, entre l'entrée et la sortie d'un labyrinthe.
On supposera que le labyrinthe est représenté par une grille (matrice) où certaines cases sont fermées. Chaque case est identifiée par ses coordonnées (ligne, colonne). Pour simplifier, les déplacements se feront suivant la verticale et l'horizontale (pas en diagonale). Donc si on est dans une case (i,j), on pourra se déplacer vers les cases (i+1,j) , (i,j+1) , (i-1,j) et (i,j-1) sous réserve qu'elles ne soient pas fermées ni déjà visitées.

La figure suivante montre un exemple de labyrinthe formé par une grille de 8x8.



2. Application du backtracking dans les arbres de jeux

On considère les jeux de stratégie entre deux joueurs (A et B). Les deux joueurs sont soumis aux mêmes règles (symétrie). Le jeu est déterministe (le hasard n'intervient pas).

Ce type de jeu peut être représenté sous forme d'une arborescence. Chaque noeud représente une configuration possible du jeu. un arc représente une transition légale entre deux configurations. La racine constitue la configuration initiale, les feuilles les configurations finales (gagné, perdu ou nul).

Dans cet arbre si par exemple un noeud J a comme fils les noeuds J1, J2 et J3, alors à partir de la configuration J il y a 3 coups possibles menant vers l'une des configurations J1, J2 ou J3. Une partie complète est donc une branche de l'arbre de jeu entre la racine et un noeud feuille.

A partir de la configuration initial du jeu (le niveau 0), l'un des joueurs commence la partie. Le jeu passe alors dans une autre configuration (un noeud du niveau 1) et c'est au tour du deuxième joueur de choisir son coup. Ensuite (dans le niveau 2) le tour revient au premier joueur, puis au niveau 3 la main revient au deuxième et ainsi de suite, alternativement, les deux joueurs s'affrontent jusqu'à la fin de la partie (atteindre une configuration feuille).

On aimerait écrire un programme permettant à la machine d'être l'un des joueurs.

a- Principe du MIN-MAX:

Dans le principe du MIN-MAX un des joueurs est appelé joueur *maximisant* (dans la suite ça sera toujours le joueur A), l'autre (le joueur B) est appelé joueur *minimisant*.

Si c'est au tour de A de jouer on dit que le niveau correspondant dans l'arbre est un niveau maximisant et inversement si c'est au tour de B de jouer le niveau est minimisant.

Pour que la machine (le joueur A ou B) puisse jouer de façon convenable à partir d'une configuration donnée, elle doit pouvoir choisir parmi les fils de la configuration courante celui qui représente le coup le plus favorable pour elle et donc le plus défavorable pour son adversaire. Pour pouvoir faire ce choix, l'algorithme du Min-Max attribue à chaque configuration de l'arbre une certaine valeur. Si la machine est le joueur maximisant, elle choisira parmi les fils de la configuration courante celui qui porte la plus grande valeur, et inversement, si c'est un joueur minimisant, le choix portera sur la plus petite des valeurs.

Comment Min-Max attribue-t-il des valeurs aux différentes configurations ?

On commence par attribuer des valeurs aux feuilles.(+1 si A gagne, -1 si A perd, 0 si nul). Il est claire que si A gagne, B perd et si A perd, B gagne. Ceux sont les règles du jeu qui déterminent si une configuration feuille représente une partie gagné, perdue ou nulle.

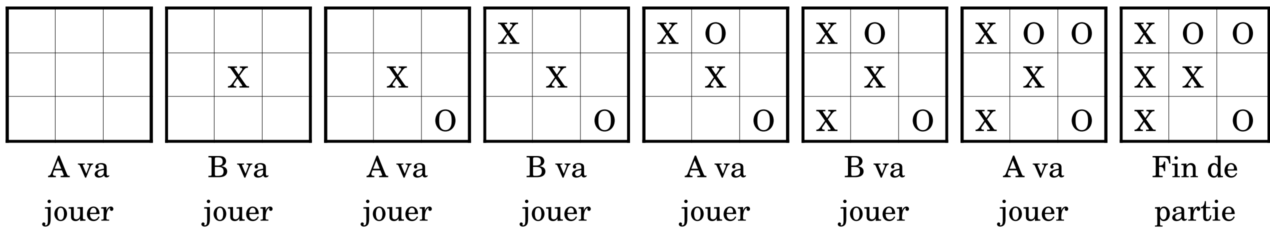
Les valeurs sont propagées vers les noeuds ascendants, jusqu'à arriver à la racine de la façon suivante:
- si c'est au tour de A de jouer, le noeud correspondant prend la plus grande des valeurs de ses fils (le coup le plus profitable pour A)
- si c'est au tour de B de jouer, le noeud correspondant prend la plus petite des valeurs de ses fils (le coup le plus profitable pour B)
Si la racine prend comme valeur 1, le joueur A peut gagner s'il ne fait pas d'erreurs. On dit que A a une stratégie gagnante.

Si la racine prend la valeur -1, le joueur A est assuré de perdre si B ne fait pas d'erreurs. Dans ce cas c'est B qui a une stratégie gagnante.
Si la racine prend la valeur 0, aucun des deux joueurs n'a de stratégie gagnante, mais tous deux peuvent s'assurer, au pire, d'un match nul en jouant aussi bien que possible. (Cas du jeu des croix et des cercles décrits plus loin).
Pour propager les valeurs de bas en haut, Min-Max parcours l'arbre des configurations avec un parcours post-ordre: avant d'évaluer un noeud donné, il faut d'abord évaluer tous ses fils. Rappelant que les parcours postordre, inordre et préordre sont des cas particuliers du parcours en profondeur.

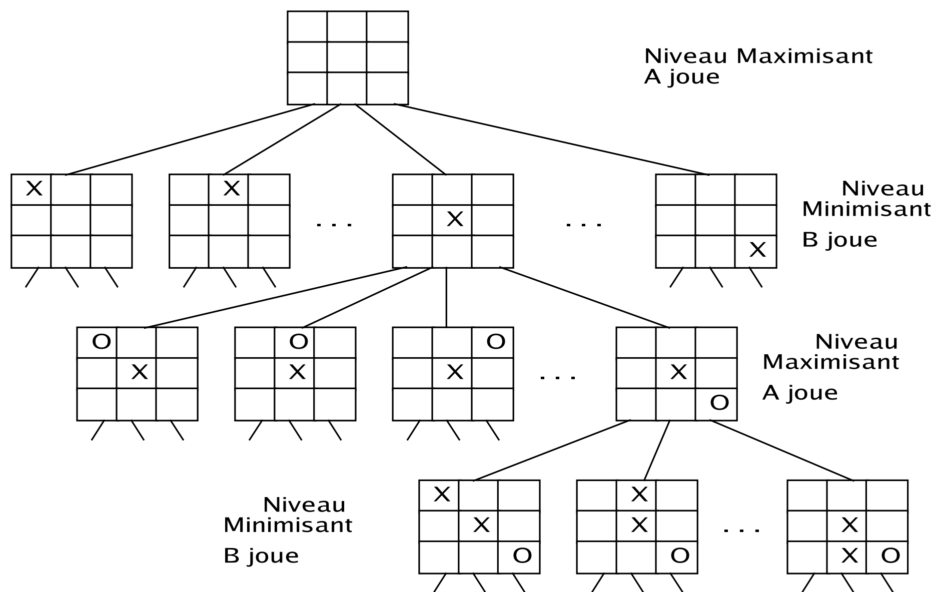
Exemples :

1. Tic-Tac-Toe (jeu des croix et des cercles)

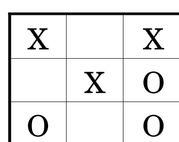
Ce jeu consiste à placer des croix et des cercles sur une grille de 9 cases (3 x 3), jusqu'à ce que l'un des joueurs aligne trois de ses symboles.
Posons que A joue avec le symbole X (les croix) et B joue avec le symbole 0 (les cercles).
A chaque coup, un joueur place un de ses symbole dans une case vide de la grille.
La configuration initiale est une grille vide.
Une configuration feuille représente une grille où il y a un alignement de 3 symboles identiques (match gagné ou perdu) ou bien il n'y a plus de cases vides dans la grille (match nul).
Voici le déroulement d'une partie en supposons que A commence à joeur en premier:



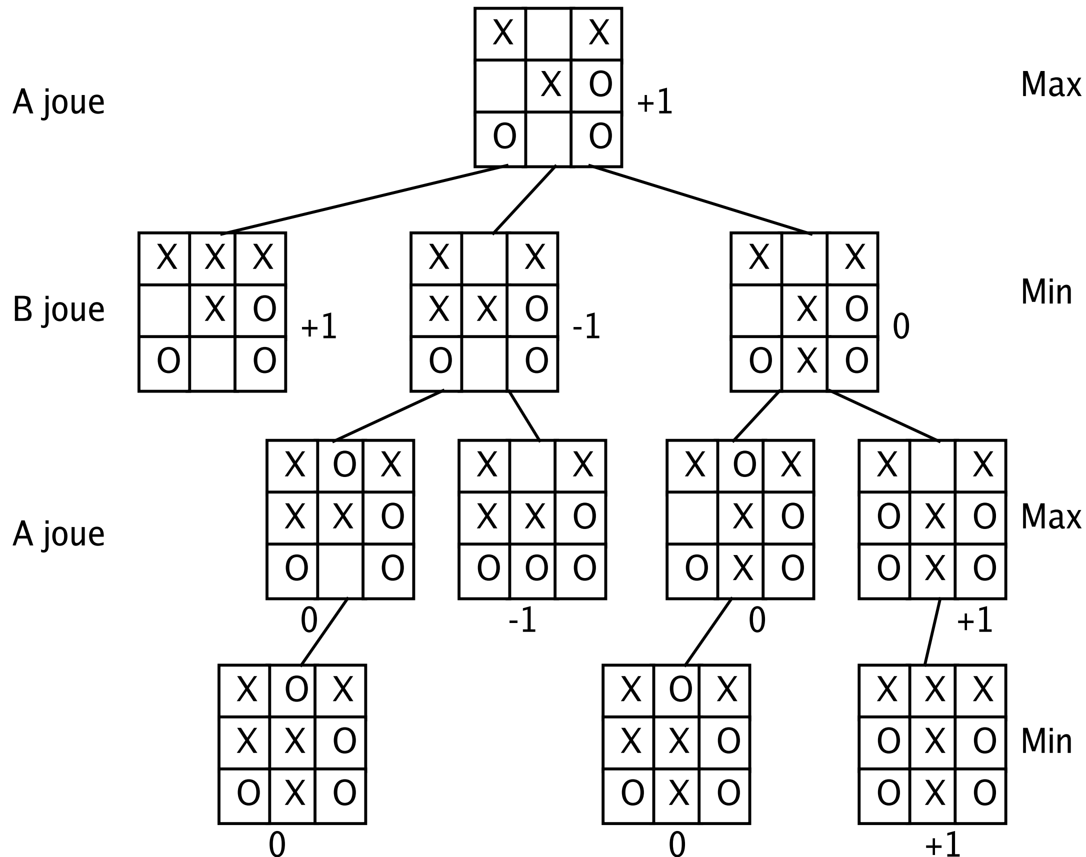
Le joueur A a gagné la partie car il a pu aligner 3 X.
Lorsque le joueur A a commencé la partie, il avait le choix entre 9 cases vide pour placer son premier X. Ensuite le joueur B avait le choix entre 8 cases vides pour placer son premier O, etc ...
La figure suivante montre les premiers niveaux de l'arbre de recherche:



Supposons qu'on est arrivé à la configuration suivante et où c'est au tour de A de jouer :



Pour évaluer cette configuration par Min-Max, il faut parcourir en postordre le sous-arbre ayant comme racine cette configuration et propager les valeurs des feuilles vers les noeuds ascendants :



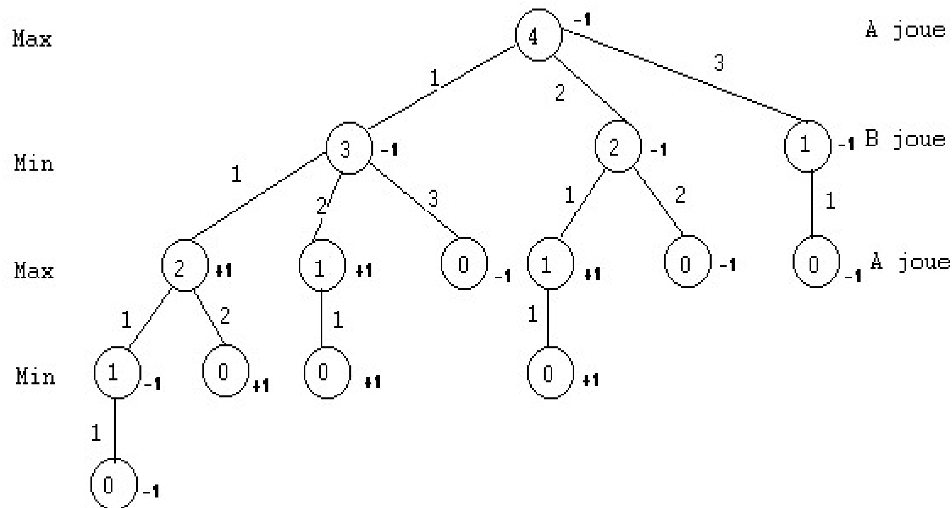
La valeur retournée par Min-Max est +1, qui veut dire que A est assuré de gagner s'il ne fait pas d'erreurs. Les noeuds étiquetés par la valeur 0 mènent vers un match nul, ceux étiquetés par -1 mènent vers une défaite de A si B ne fait pas d'erreurs.

2. Jeu de Nim:

- Initialement, il y a n billes sur une table.
- A chaque étape, Les joueurs doivent prendre un nombre de billes compris entre 1 et k (k est une constante inférieure à n fixée au début du jeu).
- Celui qui peut ramasser les dernières billes de la table remporte la partie.

Un état de l'espace de recherche peut être représenté uniquement par le nombre de billes (nb) présentes dans la table à un instant donné.
A l'état initial on a nb = n. Alors qu'un état solution est représenté par nb=0.
Prenons comme convention que A soit un joueur maximisant et B un joueur minimisant et que A commence la partie.

Pour n=4 et k=3, le joueur B a une stratégie gagnante comme le montre l'arbre suivant.



Les nombres étiquetant les arcs de l'arbre ci-dessus indiquent le nombre de billes pris par un joueur. Par exemple à partir de l'état initial (la racine nb=4) le joueur A peut prendre une, deux ou trois billes menant le jeu respectivement vers les états nb=3, nb=2 et nb=1. Ensuite B doit jouer à partir d'un de ces état, etc...
La valeur -1 au niveau de la racine de l'arbre, indique que A est assuré de perdre si B ne fait pas d'erreur.

Pour n=4 et k=2 le joueur A a une stratégie gagnante.

Algorithme du MIN-MAX :

Si l'arbre généré à partir de l'état initial n'est pas très grand (cela dépend du jeu considéré) on peut lancer une seule fois la fonction Minmax pour évaluer tous les noeuds possibles et stocker ces informations en mémoire. Ensuite lorsque le jeu se déroulera, il suffira de choisir parmi les configurations courantes celles qui portent la plus grande ou la plus petite des valeurs suivant que la machine soit un joueur maximisant, ou minimisant.
Par contre, et c'est souvent le cas, si l'arbre de recherche est trop grand pour être parcouru entièrement (le jeu d'échecs par exemple) la fonction Minmax doit être appelé au cours du déroulement du jeu, a chaque fois que la machine doit jouer. Cependant on limitera le parcours à une profondeur maximale fixée en fonction des capacités de la machine et du temps de réponse voulu.
Le problème qui se pose alors dans l'application de Minmax, est qu'une fois arrivé à la profondeur maximale, comment évaluer la configuration atteinte si ce n'est pas une feuille ?
La réponse est donnée par l'utilisation de fonctions d'estimations qui appliquées sur une configuration donnée, retournent une valeur estimée de sa qualité (proche de -1 dans le cas d'une configuartion favorable au joueur minimisant et proche de +1 pour une configuration favorable au joueur maximisant).
Ces fonctions d'estimations dépendent de chaque jeu et influent de façon importante sur la qualité de jeu de la machine.

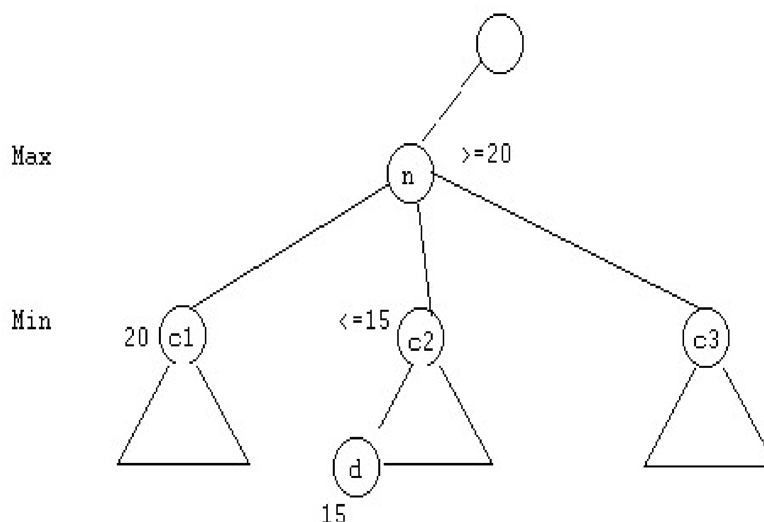
Fonction Minmax(J, Mode, Niveau) : réel
/* évalue le coût de la configuration j, sachant que si mode = Max c'est au joueur maximisant de jouer et si mode = Min c'est au joueur minimisant de jouer. La fonction retourne le coût de la configuration. */

```
Si J est une feuille :
        Retourner (coût( J ))
Sinon
        Si Niveau = 0 :
                Retourner (Estimation( J ))
        Sinon
                Si Mode = Max  :
                        Val := - infini
                Sinon
                        Val := + infini
                Fsi
                Pour chaque fils K de J:
                        Si Mode = Max
                                Val := Max(Val,Minmax(K, Min, Niveau-1))
                        Sinon
                                Val := Min(Val,Minmax(K, Max, Niveau-1))
                        Fsi
                Finpour
                Retourner Val
        Fsi
Fsi
```

A l'appel initial de Minmax, on donne la configuration que l'on veut évaluer (J), la nature du joueur (maximisant ou minimsant) qui va jouer à partir de J (Mode) et la profondeur maximale dans l'arbre que l'on veut atteindre (Niveau).
La fonction Coût(J) est appliquée uniquement à une feuille est doit retourner soit -1, soit 0 , soit +1.
La fonction Estimation(J) s'applique à n'importe quelle configuration est retourne une réel compris entre -1 et +1.
Par exemple dans le jeu d'échecs où l'arbre de jeu est trop grand pour que son remplissage depuis les feuilles jusqu'à la racine soit réalisable, chaque programme utilise une fonction d'estimation permettant d'évaluer, suivant la disposition de l'échiquier (nombre et qualité des pièces, densité de défense autour du roi, occupation des cases du centre...) la probabilité que la machine gagne à partir de cette position.

b- Principe de l'élagage Alpha-bêta (coupe de branche "pruning"):

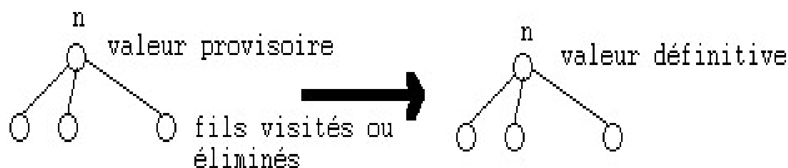
Par une simple remarque, on peut, dans l'application de Minmax, éliminer une grande partie des arbres de recherche en ignorant certains descendants d'un noeud.



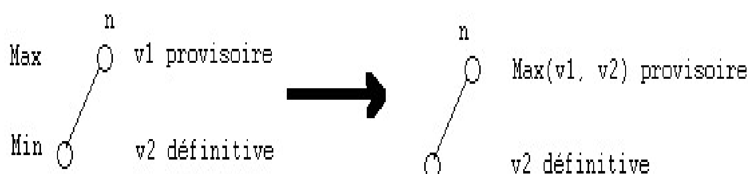
1) En parcourant tous les descendants de c1, on trouve que c1 a une valeur égale à 20.
2) Avant d'explorer les prochains fils de n, on peut déjà dire que la valeur de n est supérieure ou égale à 20. (car n est un noeud maximisant)
3) En explorant c2 et ses fils, on tombe sur un noeud d de valeur 15. Comme d est un fils de c2, on aura c2 <= 15 (noeud minimisant)
4) A ce niveau, on peut abandonner l'exploration des autres fils de c2 et passer directement à c3. Car la valeur de c2 ne peut plus influencer celle de n.

Règles de calcul :

1. Si tous les fils d'un noeud 'n' ont été examinés ou éliminés, transformer la valeur de 'n' (jusqu'ici provisoire) en une valeur définitive.

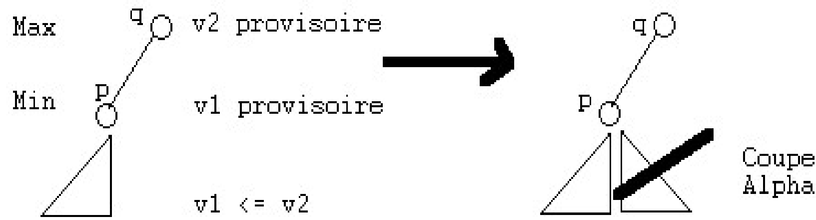


2. Si un noeud maximisant n a une valeur provisoire v1 et un fils de valeur définitive v2, donner à n la valeur provisoire Max(v1, v2). Si n est minimisant on lui donne la valeur provisoire (dans les mêmes conditions) Min(v1, v2).



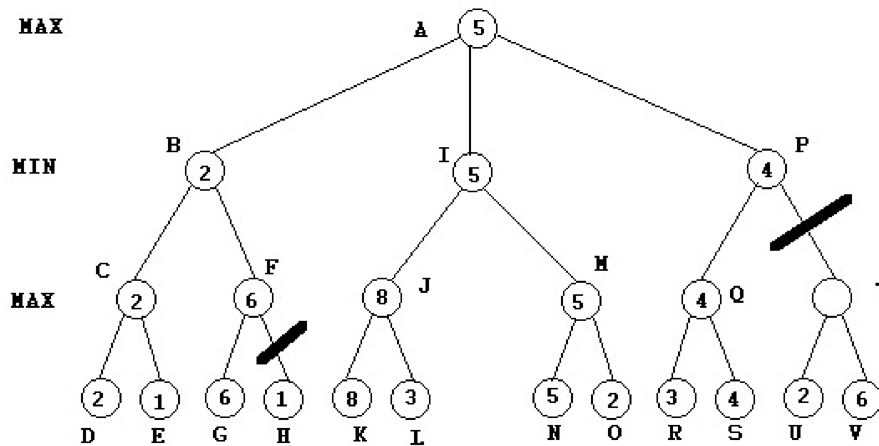
3. Si p est un noeud minimisant de père q maximisant avec des valeurs provisoires respectives v1 et v2 avec v1 <= v2, alors ignorer toute la descendance encore inexplorée de p (coupe de

type Alpha). Une coupe de type bêta est définie de manière analogue dans le cas où p est maximisant et q minimisant et v1 >= v2.



Exemple :

Soit l'arbre suivant, où on applique Minmax avec élagage alpha-bêta (les valeurs sont représentées à l'intérieur des noeuds) :



<u>Scénario :</u>
Noeud visité en postordre : D avec valeur définitive 2 (R2 : V(C) = 2 provisoire)
Noeud visité E avec valeur définitive 1 (R1 : V(C) = 2 définitive - R2 : V(B) = 2 provisoire)
Noeud visité G de valeur 6
(R2 : V(F) = 6 - R3 : Coupe Béta sur la branche H - R2 : V(A) = 2 provisoire)
Noeud visité K de valeur 8 (R2 : V(J) = 8 provisoire)
Noeud visité L de valeur 3 (R1 : V(j) = 8 définitive - R2 : V(I) = 8 provisoire)
Noeud visité N de valeur 5 (R2 : V(M) = 5 provisoire)
Noeud visité O de valeur 2
(R1 : V(M) = 5 définitive - R1 : V(I) = 5 définitive - R2 : V(A) = 5 provisoire)
Noeud visité R de valeur 3 (R2 : V(O) = 3)
Noeud visité S de valeur 4
(R1 : V(Q) = 4 - R2 : V(P) = 4 provisoire - R3 : Coupe Alpha sur branche TUV)

C. Recherche en largeur (Breadth First Search)

La recherche en largeur est une technique générale de recherche de solution dans un graphe

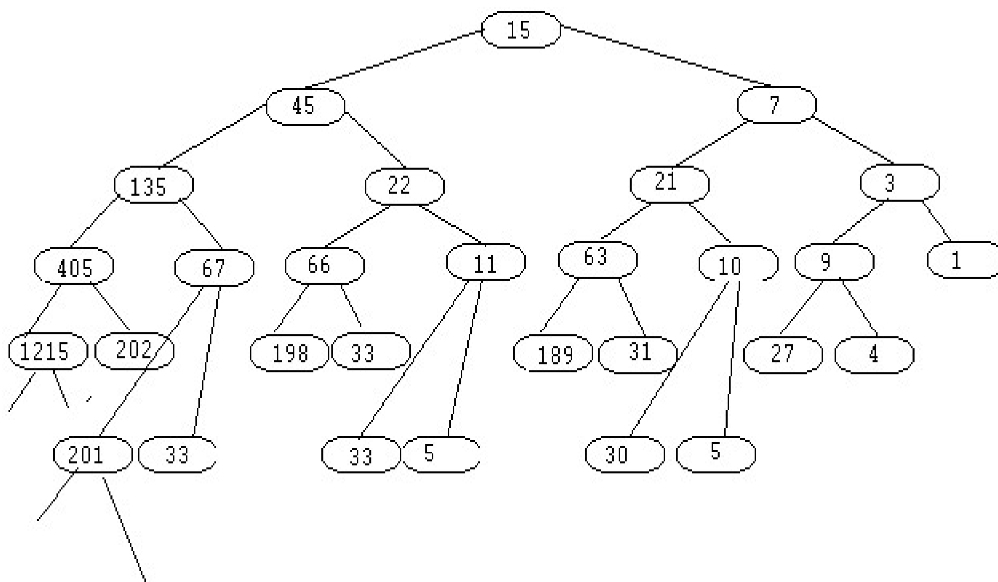
implicite où le parcours est en largeur.
On rappelle que le parcours en largeur progresse dans toutes les directions en même temps ce qui le fait consommer beaucoup d'espace mémoire. Par contre c'est l'état solution le plus proche de l'état initial qui sera trouvé le premier, même s'il y a des branches infinies.

Exemple :
On voudrais passer du nombre 15 au nombre 4 en utilisant uniquement les fonctions f et g suivantes : f(x) = 3x et g(x) = x div 2

L'état initial est représenté par le nombre 15, l'état final est représenté par le nombre 4. A partir d'un état quelconque on peut appliquer les fonction f et g menant vers deux autres états.

L'espace de recherche est alors un graphe infini, donc c'est la recherche en largeur qui est recommandée, en plus on trouvera le nombre minimal d'application des fonctions f et g pour passer de 15 à 4.

Prenons comme convention, dans le graphe suivant, que les arcs sortant à gauche représentent les applications de la fonction f et ceux sortant à droite représentent les application de g :



La solution est donc :

4 = g(f(g(g(15))))