CHAPITRE 4: OPTIMISATION COMBINATOIRE, MÉTHODES EXACTES

COURS : COMPLEXITÉ ET OPTIMISATION

RÉALISÉ Par : DR. S. SLATNIA

UNIVERSITÉ DE BISKRA
MASTER D'INFORMATIQUE
2021-2022

DÉFINITION D'OPTIMISATION

- L'optimisation vise à résoudre des problèmes où l'on cherche à déterminer parmi un grand nombre de solutions candidates celle qui donne le meilleur rendement.
- On cherche à trouver une solution satisfaisant un ensemble de contraintes qui minimise ou maximise une fonction donnée.
- L'application de l'optimisation est en expansion croissante et se retrouve dans plusieurs domaines.

$$\min_{x \in R^n} f(x) \quad \text{s.c. } x \in S \qquad (1)$$

  - où $x = (x_1, x_2, \ldots, x_n)^T$ est un vecteur de R^n,
  - $f : R^n \rightarrow R$ est la fonction que l'on désire minimiser (appelée fonction objectif),
  - $S \subseteq R^n$ est l'ensemble dans lequel les points doivent appartenir, et s.c. est l'abbréviation de sous la ou les contraintes.

- La formulation (1) signifie que l'on cherche à trouver une solution du domaine réalisable $x^* \in S$ dont la valeur de la fonction objectif est la plus petite.

DÉFINITION D'OPTIMISATION

Définition 1. Une solution $x^* \in S$ est un minimum global de la fonction f sur le domaine S si

$$f(x^*) \leq f(x) \qquad \forall x \in S.$$

La valeur optimale est $f(x^*)$.

- NB. Le minimum global n'est pas nécessairement unique, mais la valeur optimale l'est.

Définition 2. Une solution $x^* \in S$ est un minimum local de la fonction f sur le domaine S si

$$f(x^*) \leq f(x) \qquad \forall x \in S \cap B\varepsilon(x^*).$$

Les maxima sont définis de façon similaire (remplacer ≤ par ≥).

DÉFINITION D'OPTIMISATION

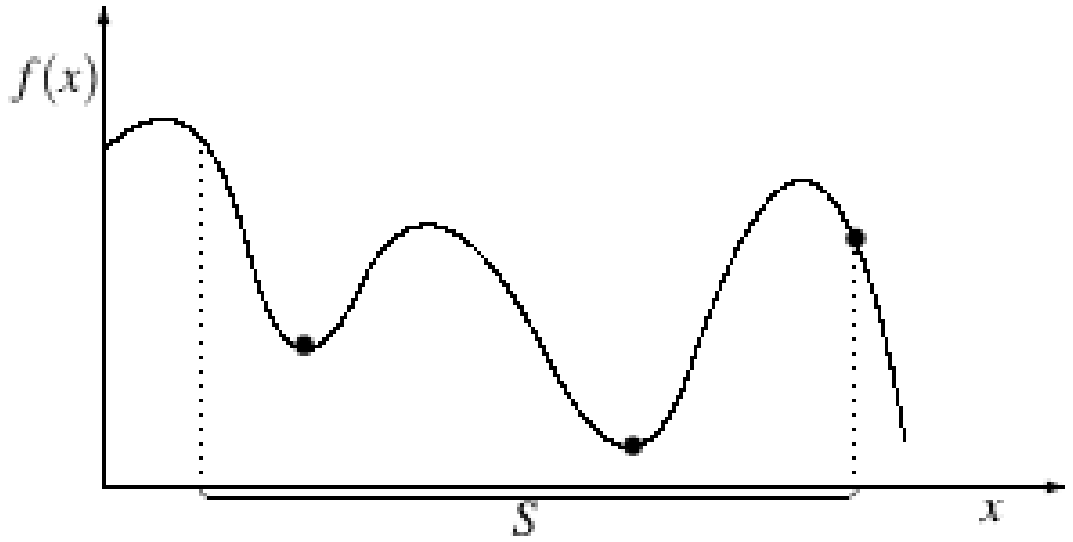

Fig. Trois minima locaux, dont un global

PROBLÈME D'OPTIMISATION

- **Problème d'optimisation**

Un problème d'optimisation est un problème pour lequel on veut trouver non seulement une solution, mais la meilleure solution.

- **Un problème d'optimisation discret**

Est un problème dont les variables de décisions appartiennent à des domaines discrets. On parle de programmation entière si les variables sont entières.

- **Un problème d'optimisation combinatoire**

Est un problème de choix de la meilleure combinaison parmi toutes les combinaisons possible.

La plupart des problèmes combinatoires sont formulés comme des programmes entiers.

LA CLASSIFICATION DES PROBLÈMES D'OPTIMISATION

On peut classer les différents problèmes d'optimisation que l'on rencontre dans la vie courante en fonction de leurs caractéristiques [34]:

1. Nombre de variables de décision :

– Une⇒ monovariable.

– Plusieurs⇒ multivariable.

2. Type de la variable de décision :

– Nombre réel continu⇒ continu.

– Nombre entier⇒ entier ou discret.

– Permutation sur un ensemble fini de nombres⇒ combinatoire.

LA CLASSIFICATION DES PROBLÈMES D'OPTIMISATION

3. Type de la fonction objectif :

– Fonction linéaire des variables de décision⇒ linéaire.

– Fonction quadratique des variables de décision⇒ quadratique.

– Fonction non linéaire des variables de décision⇒ non linéaire.

4. Formulation du problème :

– Avec des contraintes⇒ contraint.

– Sans contraintes⇒ non contraint.

BRANCH AND BOUND

BRANCH AND BOUND [33]

- Un algorithme par **séparation et évaluation**, ou ***branch and bound*** en anglais, est une méthode générique de résolution de problèmes **d'optimisation combinatoire**.
- L'optimisation combinatoire consiste à trouver un point **minimisant** une fonction, appelée coût, dans un **ensemble dénombrable** [33].
- Une méthode naïve pour résoudre ce problème est d'énumérer toutes les solutions du problème, de calculer le coût pour chacune, puis de donner le minimum.
  - Parfois [33], il est possible d'éviter d'énumérer des solutions dont on sait, par l'analyse des propriétés du problème, que ce sont de mauvaises solutions (des solutions qui ne peuvent pas être le minimum).
    - La méthode *séparation et évaluation* est une méthode générale pour cela.
- Cette méthode est très utilisée pour résoudre des **problèmes NP-complets** [33].

BRANCH AND BOUND [33]

- **Problème d'Optimisation**

  Soit S un ensemble fini mais de grande cardinalité qu'on appelle ensemble (ou espace) des solutions réalisables. On dispose d'une fonction f qui pour toute solution réalisable x de S, renvoie à un coût $f(x)$. Le but du problème est de trouver la solution réalisable x de **coût minimal**.

  ✓ le problème est trivial : une telle solution x existe bien car l'ensemble S est fini.

- **Difficultés**

  1. N'existe pas forcément un algorithme **simple** pour énumérer les éléments de S.
  2. Le nombre de solutions réalisables est très grand,
     - Le temps d'énumération de toutes les solutions est **impossible**
       - La **complexité en temps** est **exponentielle**.

- **Solution**

  Les méthodes par séparation et évaluation,

  - La **séparation** permet d'obtenir une méthode générique pour énumérer toutes les solutions
  - L'**évaluation** évite l'énumération systématique de toutes les solutions.

BRANCH AND BOUND [33]

- **Separation**

  La phase de séparation consiste à diviser le problème en un certain nombre de sous-problèmes qui ont chacun leur ensemble de solutions réalisables, et que tous ces ensembles forment une **partition** de l'ensemble *S*. Ainsi, en résolvant tous les sous-problèmes et en prenant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial.

  - Ce principe de séparation peut être appliqué de manière récursive à chacun des sous-ensembles de solutions obtenus,.

  - Les ensembles de solutions (et leurs sous-problèmes associés) ainsi construits ont une hiérarchie naturelle en arbre, souvent appelée *arbre de recherche* ou *arbre de décision*.

- **Évaluation**

- L'évaluation d'un nœud de l'arbre de recherche a pour but de déterminer l'optimum de l'ensemble des solutions réalisables associé au nœud en question ou, au qu'il n'y a pas de solution optimale. Lorsqu'un tel nœud est identifié dans l'arbre de recherche, il est donc inutile d'effectuer la séparation de son espace de solutions.

- À un nœud donné, l'optimum du sous-problème peut être déterminé lorsque le sous-problème devient « suffisamment simple ».

- Pour déterminer qu'un ensemble de solutions réalisables ne contient pas de solution optimale, la méthode la plus générale consiste à déterminer un minorant du coût des solutions contenues dans l'ensemble Si on arrive à trouver un minorant qui est supérieur au coût de la meilleure solution trouvée jusqu'à présent, on a alors l'assurance que le sous-ensemble ne contient pas l'optimum.

L'algorithme [25]

C'est un type de parcours en largeur où on garde dans la file les chemins générés.

```
CreerFile( F );                          /* une file de priorité ordonnée par f */
Z := {Racine};                           /* un chemin formé par un seul sommet */
Enfiler( F , <Z,f(Racine)> );
Trouv := FAUX;
TantQue Non FileVide(F) et Non Trouv
      Defiler( F , Z );
      Soit x le dernier sommet du chemin Z;
      Si x <> BUT
         Soient y1,...yk les succ de x qui n'appartiennent pas à Z;
         Pour i:=1,k
              T := Z + {yi}       /* former de nouveaux chemins */
              Enfiler(F,<T,f(yi)>);     /* et les enfiler */
         FP
         etiq: /* pour le moment rien à faire ici */
      Sinon
            Trouv := VRAI
       Fsi
FTQ
```

❖ Dans cet algorithme, la file contient des chemins issus de la racine, à chaque étape, on défile le chemin le plus prioritaire et on l'étend par les différentes alternatives présentes au niveau du dernier sommet du chemin. Les nouveaux chemins ainsi construit sont rajoutés à la file [25].

❖ Pour ordonner les chemins dans la file, on utilise la fonction d'estimation f(x) où x est le dernier sommet du chemin. Dans les méthodes Branch & Bound ou bien dans A*, cette fonction d'estimation est décomposée en deux partie:

$f(x) = g(x) + h^*(x)$

- g(x) représente le coût du chemin entre la racine est x et
- h*(x) représente une estimation du coût entre x et un sommet solution (but) accessible a partir de x.
- f(x) estimation du coût d'un chemin entre la Racine et le BUT passant par x.

❖ Pour que le chemin trouvé soit toujours optimal, on impose à h*(x) de ne jamais surestimer le coût du trajet restant réel. On dit que h*(x) est une fonction de sousestimation.

- ➢ si $h^*(x) = v$ alors le coût réel du trajet entre x et le BUT est forcément $\geq v$ [25].

✓ Si h*(x) = 0 quelque soit x, la méthode est appelée « Branch and Bound » pure.
✓ Si h*(x) est une fonction de sousestimation du trajet restant entre x et le BUT, la méthode est appelée « Branch and Bound » avec sousestimation.

❖ Comme le problème du plus court chemin vérifie le principe d'optimalitéé (dans un chemin optimal, tout souschemin doit aussi être optimal), on peut réorganiser la file de priorité de l'algorithme de base pour éliminer tous les souschemins menant vers un même sommet pour ne garder que le plus court d'entre eux.

- ➢ application de la programmation dynamique. Ceci permettra de rendre l'algorithme encore plus efficace.
- ➢ Cette variante est appelée « Branch and Bound » avec programmation dynamique.

❑ S'il existe dans F plusieurs chemins menant vers un même sommet, alors supprimer tous les sous chemins inutiles (menant vers le même sommet et de coût plus grand) [25].

PROGRAMMATION DYNAMIQUE

RECHERCHE ARBORESCENTE, L'EXPLORATION SANS INFORMATION

STRUCTURES DE DONNÉES POUR LES GRAPHES

- Plusieurs structures de données sont utilisées, notamment matrices ou listes.
- La **complexité des algorithmes** dépendra de **la représentation choisie**.
- Soit G =(V,E) un graphe non orienté, où
  - V est l'ensemble des sommets et E l'ensemble des arêtes ,
  - Le nombre de sommets est **n** =|V|,
  - Le nombre d'arêtes est **m**=|E|
    - **$m=O(n^2)$**

PARCOURS DE GRAPHE

- Il existe essentiellement deux méthodes de parcours de graphes. Chacune d'elles utilise la notion d'arborescence (organisation hiérarchique).

- Pour parcourir un graphe, on va produire un recouvrement du graphe par une arborescence, ou plusieurs.

❑ Parcours en largeur

On se donne une racine r et on cherche à l'aide d'une pile une arborescence des plus courts chemins à partir de r :

Sommets x tels qu'il existe **un unique** chemin de r à x.

Complexité

- Chaque sommet est empilé/dépilé au plus une fois,
- Chaque arc est exploré au plus une fois.

→ Parcours en temps **O(n+m).**

❑ Parcours en **profondeur**

- Donner une racine r, et déplacer dans l'espace de recherche en suivant un parcours en profondeur du graphe à partir de r 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.

Complexité

❖ On visite encore tous les sommets accessibles depuis la racine.

❖ La version non récursive n'est pas optimisée, car on parcourt plusieurs fois la liste des successeurs :

➢ il faudrait empiler le rang du successeur visité en même temps qu'on l'empile, et incrémenter ce rang au moment où on le dépile.

→ Complexité en **O(n+m).**

❑ Backtracking (retour sur trace) [98]

- **Le retour sur trace (appelé aussi backtracking en anglais)**

Est un algorithme qui consiste à revenir légèrement en arrière sur des décisions prises afin de sortir d'un blocage.

- La résolution d'un problème par la méthode de backtracking repose sur la construction d'une solution partielle que l'on va améliorer afin de s'approcher de plus en plus de la solution finale.

- Si une solution partielle ne peut pas être améliorée, elle est **abandonnée** et l'on revient en arrière pour examiner d'autres solutions possibles.

- Dans un algorithme de backtraking, on explore toutes les solutions possibles, quand une solution possible se termine en impasse, on revient en arrière et on teste d'autres solutions.

❑ Backtracking (retour sur trace)

Principe

Dans l'utilisation du backtracking pour résoudre un problème particulier, nous avons besoin de deux choses :

1. Une procédure pour examiner une solution partielle (notée SP par la suite) afin de déterminer si :
   - Il s'agit d'une solution actuelle **ACCEPTABLE**.
   - Il s'agit d'une solution actuelle à **ABANDONNER** (elle ne respecte pas la règle du jeu).
   - Il faut poursuivre **L'EXTENSION** de la solution actuelle.

2. Une procédure pour ETENDRE la SP, générant une ou plusieurs solutions qui se rapprochent de la solution finale.

❑ 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.
  → **Un 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).

❑ Principe du MIN-MAX

- Dans le principe du MIN-MAX un des joueurs est appelé
  - Joueur *maximisant* = le joueur A,
  - Joueur *minimisant* = le joueur B.

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

→ Choisir parmi les fils de la configuration courante celui qui représente la situation la plus favorable pour elle
  - ➢ 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.

❑ Principe du MIN-MAX

Comment Min-Max attribue des valeurs aux différentes configurations ?

- On commence par attribuer des valeurs aux feuilles.
  - ✓ **(+1)** si A gagne,
  - ✓ **(1)** si A perd,
  - ✓ **0** si Nul

- 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.
  - ✓ Si c'est au tour de **B** de jouer, le noeud correspondant prend la plus **petite** des valeurs de ses fils.

❑ Principe du MIN-MAX

→ Si la racine prend comme valeur 1, le joueur A peut gagner s'il ne fait pas d'erreurs. On dit que A à 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.

❖ Pour propager les valeurs de bas en haut, Min-Max parcours l'arbre des configurations avec un parcours **postordre**: avant d'évaluer *un noeud* donné, il faut d'abord évaluer *tous ses fils*.

Exemple : TicTacToe (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 (croix) et B joue avec le symbole 0 (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 bien il n'y a plus de cases vides dans la grille.

- Voici le déroulement d'une partie en supposons que A commence à jouer en premier:

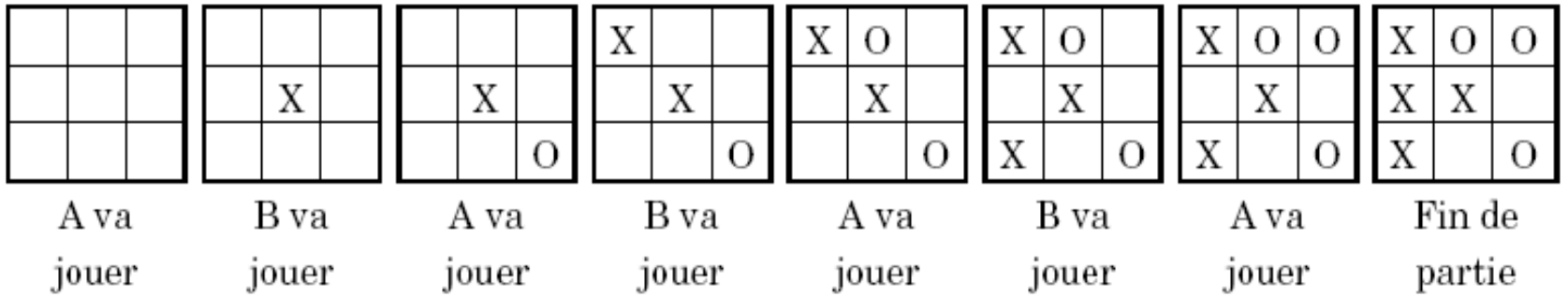

Figure. Exemple d'un jeu des croix et des cercles

- 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:

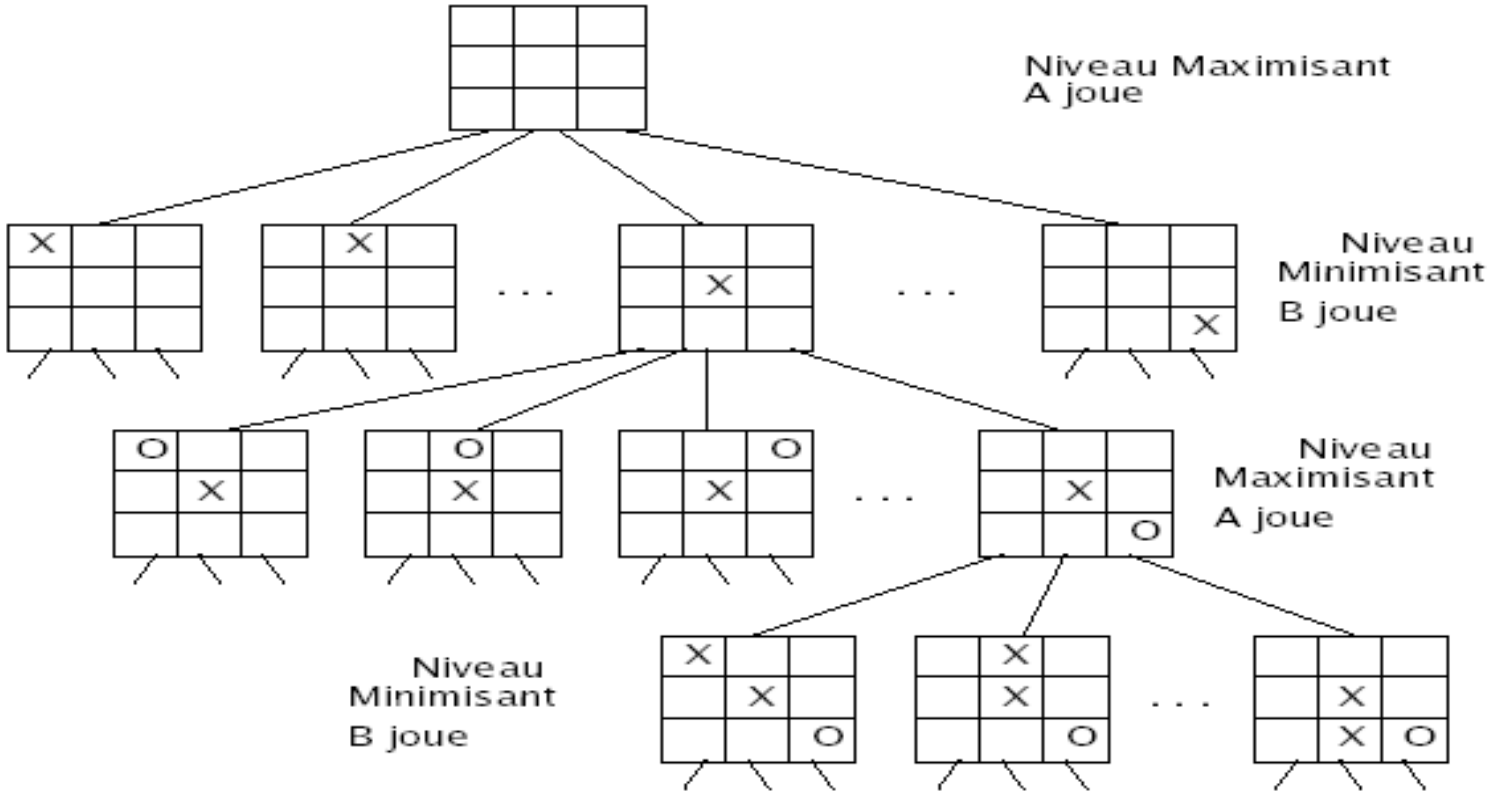

Figure. Arbre des niveaux de jeu

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

X		X
	X	O
O		O

Figure. Exemple de configuration de jeu

- Pour évaluer cette configuration par Min-Max, il faut parcourir en **post** ordre le sous arbre ayant comme racine cette configuration et propager les valeurs des feuilles vers les nœuds 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 **0** = match nul,
  1 = une défaite de A si B ne fait pas d'erreurs.

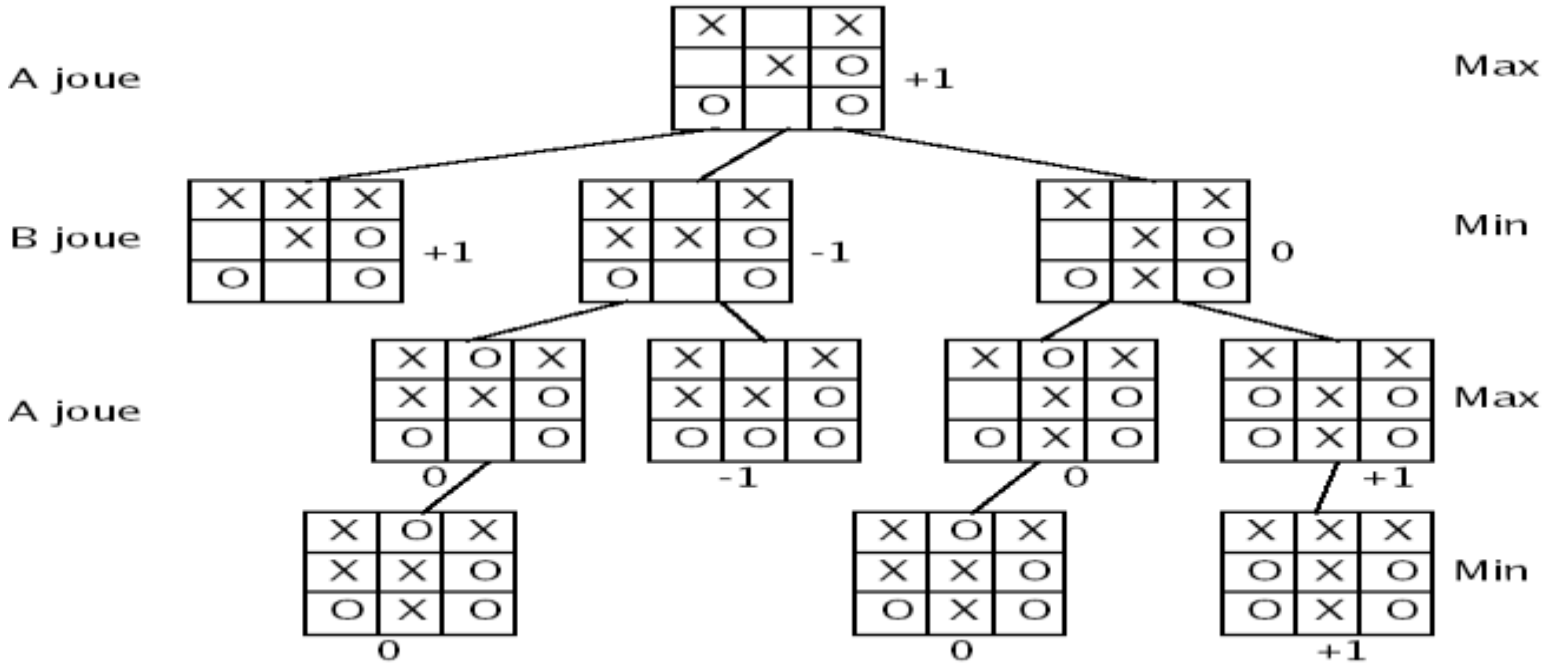

Figure. Arbre des niveaux de joue de la configuration de la figure précédente.

Exercice 1 – les n reines

Il faut placer 8 reines dans un échiquier (matrice 8X8) sans qu'aucune d'entres elles ne soient 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.

1. Placer les 4 reines sur un tableau 4 x 4 en utilisant l'algorithme backtracking du cours, montrez toutes les configurations considérées lors de son fonctionnement.
2. Donner l'algorithme qui permet de placer les 8 reines dans l'échiquier.

Correction 1 – *Les 8 reines*

1. Une configuration consiste à placer les reines sur les colonnes de 1 à k (k=0..4). On la représente par un vecteur de k entiers qui correspondent aux lignes de ces reines

1
13
14
142
2
24
241
2413- trouvé

Correction 1 – *Les 8 reines*

2. Procedure Placer (K , Col, DiagP, DiagN)

{On a déjà réussi à placer K reines et on place les reines suivantes,}

```
debut
  Si (K = 8) alors Ecrire (V)
  Sinon
    Pour j = 1,8 :
      Si (j Col ET k+1 + j DiagP ET k+1 j DiagN) alors /* 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
Fin.
```

/* Col, DiagP et DiagN sont les ensembles de cases déjà condamnées.*/

Avec Placer(0, {}, {}, {}) comme appel initial.

Col, DiagP et DiagN sont initialement vides {}.

Exercice 2 – *problèmes 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) > (3Q(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) > (4Q(A)).
7. on peut verser tout le contenu de A dans B Si A est non vide.
8. on peut verser tout le contenu de B dans A Si B est non vide.

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.

- En considérant, l'état initial est (0,0), donner un algorithme de recherche en profondeur pour arriver à les états solution ont la forme : (2, n) avec n quelconque.

Correction 2 – *problèmes cruches d'eau*.

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):