UNIVERSITE MOHAMED KHIDER – BISKRA
FACULTE DES SCIENCES EXACTES, DES SCIENCES DE LA NATURE ET DE LA VIE
DEPARTEMENT D'INFORMATIQUE

Complexité et Optimisation

Support de cours et TD

Master d'informatique

Réalisé par : Dr. S. SLATNIA

Version amélioée

05/01/2016

Chapitre 4.

OPTIMISATION COMBINATOIRE, METHODES EXACTES

1. Définitions

1.1. 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. Plus précisément, 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 [33].

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

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 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 ≥).

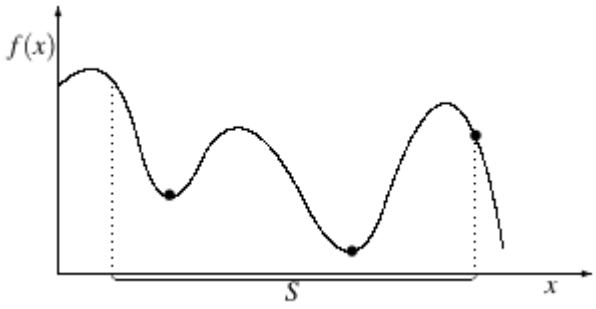

Fig 1. Trois minima locaux, dont un global

1.2. Définition de 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.

1.3. 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.

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.

2. Méthodes exactes

2.1. Branch and bound [25]

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].

2.1.1. 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.

2.1.1. 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.

2.1.3. L'algorithme

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 présente s au niveau du dernier sommet du chemin. Les nouveaux chemins ainsi construit sont rajoutés à la file.

❖ Pour ordonner les chemins dans la file, on utilise la fonction d'estimation f(x) où x est ledernier 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 lecoût du chemin entre la racine est x. h*(x) représente une estimation du coût restant entre x et un éventuel sommet soluti on (BUT) accessible à partir de x.

  - f(x) serait une 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 sur estimer le coût du trajet restant réel. On dit que h*(x) est une fonction de sous-estimation.

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

✓ Si h*(x) = 0 quelque soit x, la méthode est appelée « Branch and Bound » pure.

✓ Si h*(x) est une fonction de sous-estimation 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 réo

rganiser la file de priorité de l'algorithme de base (au niveau de l'étiquette etiq) pour éliminer tous les sous
chemins menant vers un même sommet pour ne garder que le plus court d'entre eux. C'est une
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].

2.2. Recherche arborescente, l'exploration sans information

1. Structures de données pour les graphes

Pour représenter un graphe en machine, plusieurs structures de données sont utilisées, notamment matrices ou listes de successeurs. 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)).

Définitions 1.
On rappelle quelques définitions élémentaires :

- **Degrés** : Le degré δ(v) d'un sommet v∈V est le nombre d'arêtes de E incidentes en v. Le degré Δ(G) du graphe G est le maximum des degrés d'un sommet.

- **Chemin** : Soient u, v deux sommets de V. Un chemin de u à v est un ensemble d'arêtes $e_1,e_2,...,e_k$ de E telles que e_i =(v_{i-1},v_i), v_0=u et v_k =v. Si tous les vi sont distincts, le chemin est élémentaire. Si u =v, le chemin élémentaire est un cycle élémentaire.

- **Composantes connexes** : Une composante connexe est une classe d'équivalence de la relation R définie sur V×V par uRv si et seulement s'il existe un chemin de u à v. Le graphe G est connexe si tous les sommets sont dans la même composante connexe.

- **Sous-graphe** : Un sous-graphe de G est un graphe (W,F) où W⊂V, T⊂E, et toutes les arêtes de F relient deux sommets de W. Le sous-graphe induit par un sous-ensemble de sommets comprend toutes les arêtes qui ont leurs deux extrémités dans ce sous-ensemble. De même pour le sous-graphe induit par un sous-ensemble d'arêtes.

- **Arbre** : Un arbre est un graphe connexe sans cycle**.**

2. Parcours de graphe

Il existe essentiellement deux méthodes de parcours de graphes. Chacune d'elles utilise la notion d'arborescence : pour parcourir un graphe, on va en effet produire un recouvrement du graphe par une arborescence, ou plusieurs si le graphe n'est pas connexe.
– G=(V,E) graphe orienté à n =|V| sommets et m=|E| arêtes.

2.1. 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).

2.2. Parcours en profondeur

La technique consiste à 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. Nous utilisons la récursive naturelle (utilisation d'une pile).
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).

3. Backtracking (retour sur trace)

- 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.

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.

3.1.1. 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).

3.1.1.1. 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.

→ 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.

Comment Min-Max attribuetil 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.

  → 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 (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 bien il n'y a plus de cases vides dans la grille.

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

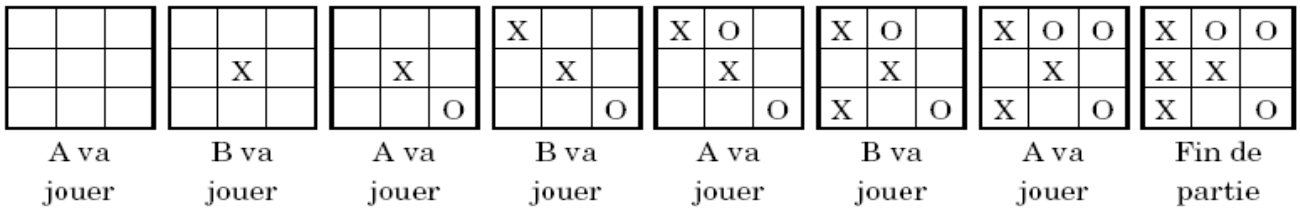

Figure 11. Exemple d'un jeu des croix et des cercles

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:

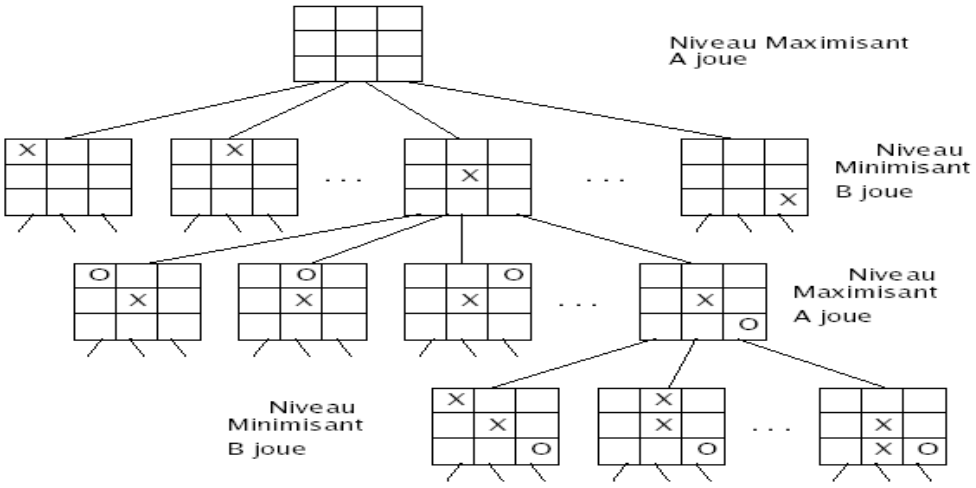

Figure 12. Arbre des niveaux de joue

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

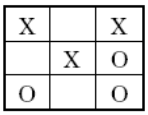

Figure 13. Exemple de configuration de joue

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 :

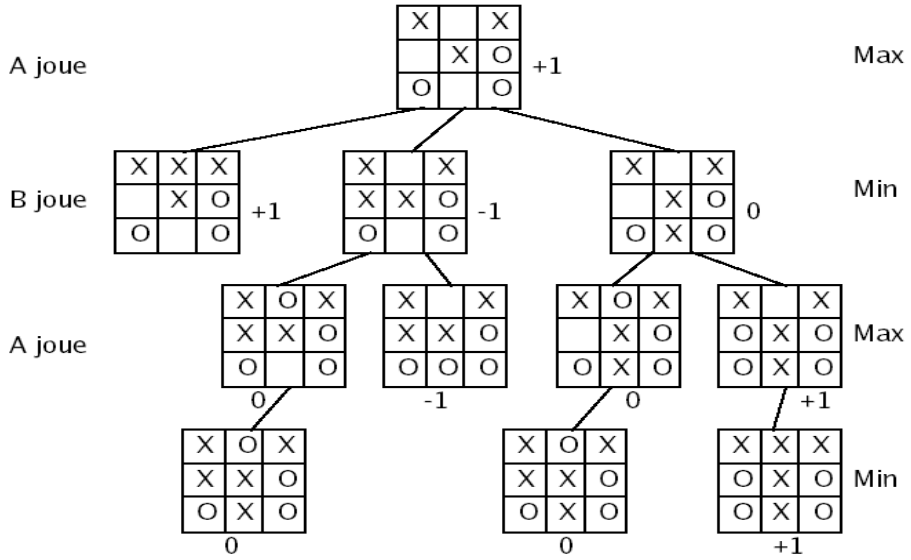

Figure 14. Arbre des niveaux de joue de la configuration de la figure 10.

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.