Complexité (Quatrième année)

TD8

Maud MARCHAL

18 avril 2011

Plan

Introduction

- **Problème d'optimisation** : rechercher un élément optimum dans un ensemble combinatoire de solutions possibles.
- On a un ensemble fini E très grand (cardinal n) et on peut effectuer l'arborescence des solutions possibles à rechercher. Les solutions possibles sont :
  - Enumération de tous les éléments de E.
  - Subdivision de E dans n' sous-ensembles non-vides.
  - Réitération de la subdivision jusqu'à ce que tous les sous-ensembles ne contiennent plus qu'un seul élément.
- Exemple : ensemble des n ! permutations de n éléments : représentation sous forme d'arbres des solutions.
- Dans la suite de ce cours, nous allons présenter les **méthodes faibles** : elles sont basées sur des techniques de parcours de graphe et peuvent être adaptées à de nombreux domaines.

Exemples de problèmes classiques

- Les explorateurs et les cannibales.
  Trois cannibales et trois explorateurs doivent traverser un fleuve en empruntant une barque qui ne peut contenir que deux personnes. Or, si sur une des deux rives, à un moment quelconque, on trouve plus (strictement) de cannibales que d'explorateurs, lers premiers font chauffer la marmite. Les explorateurs doivent donc trouver une méthode pour qu'à aucun moment ils ne soient en infériorité numérique.
  Un état peut être représenté par un triplet (X,Y,P) avec P={D,G}, X le nombre d'explorateurs et Y le nombre de cannibales. L'état initial est (3,3,G) et on cherche à atteindre (0,0,D).
- Le jeu d'échecs.
- Le problème du voyageur de commerce.
- Le jeu de poker.

Plan

Introduction

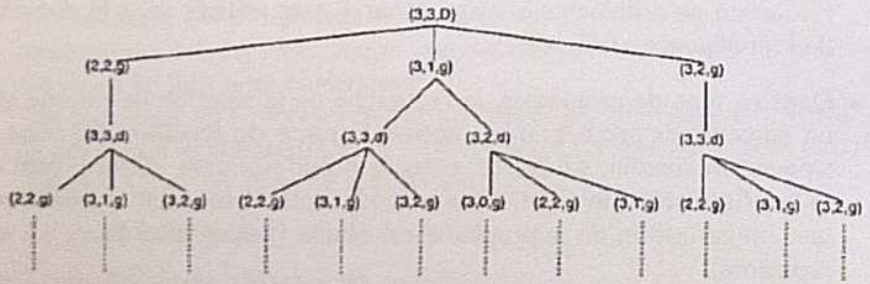
- Les problèmes d'optimisation peuvent être classiquement résolus à l'aide de techniques de recherche basée sur l'exploration d'arbres et de graphes.
- Les techniques d'exploration de graphes sont des algorithmes de résolution de problèmes à usage général, très utilisés dans le domaine de l'intelligence artificielle.
- Dans ce type de problèmes, la recherche de la solution se résume en un parcours de arbre/graphe (appelé **Espace de Recherche**) pour trouver un chemin, s'il existe, entre le nœud/sommet initial (l'**état ou la configuration initiale**) et un nœud/sommet solution représentant une configuration où le problème est résolu (l'**état final** du problème).
- Le passage d'un état à un autre est réalisé par l'application de règles définissant les propriétés et caractéristiques du problème à résoudre.

Représentation formelle sous forme d'arbres

- La méthode de représentation par arbre consiste, à chaque fois que l'on génère un état, à le relier simplement à son prédecesseur sans faire aucun test d'occurence : on ne vérifie pas si l'état a déjà été généré.
- Avantage : rapidité (test d'occurence coûteux)
- Inconvénients :
  - Duplication d'état (problème mémoire)
  - Temps d'exécution long

Représentation formelle sous forme d'arbres

Exemple en utilisant un arbre pour le problème des explorateurs et des cannibales :

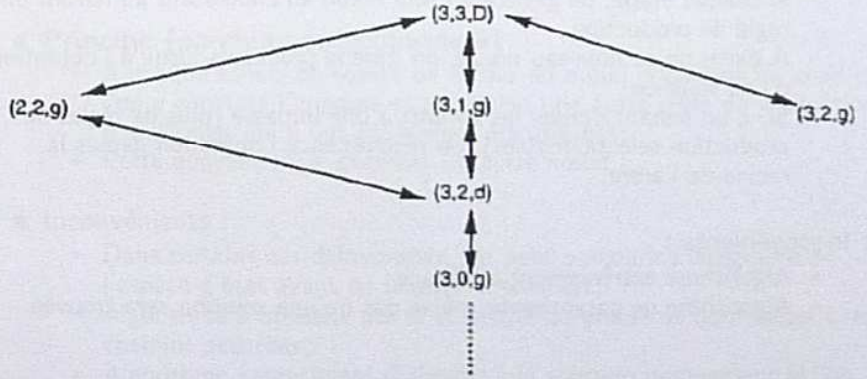


Représentation formelle sous forme de graphes

- La méthode des graphes est similaire à celle des arbres.
- Chaque nouvel état généré est stocké mais on effectue avant le stockage un test d'occurence : si l'état a déjà été généré, on se contente de construire un arc indiquant le rebouclage.
- Avantage et inconvénients : inverses à ceux des arbres.

Représentation formelle sous forme de graphes

Exemple en utilisant un graphe pour le problème des explorateurs et des cannibales :



Les différentes stratégies de résolution

- Algorithme du British Museum
- Recherche en profondeur et retour-arrière (ou recherche par états successifs)
- Recherche en largeur
- Recherche par Heuristique (dernière partie du cours)

Algorithme du British Museum

- **Principe** :
  - A chaque étape, on génère un seul nœud en choisissant au hasard une règle de production ;
  - A partir de ce nouveau nœud, on itère le processus jusqu'à l'obtention d'une solution ;
  - Si, à un instant donné, on aboutit à une impasse (plus de règles de production sélectionnables), on recommence l'opération depuis la racine de l'arbre.
- Inconvénients :
  - Algorithme extrêmement inefficace ;
  - Algorithme ne garantissant même pas qu'une solution sera trouvée.

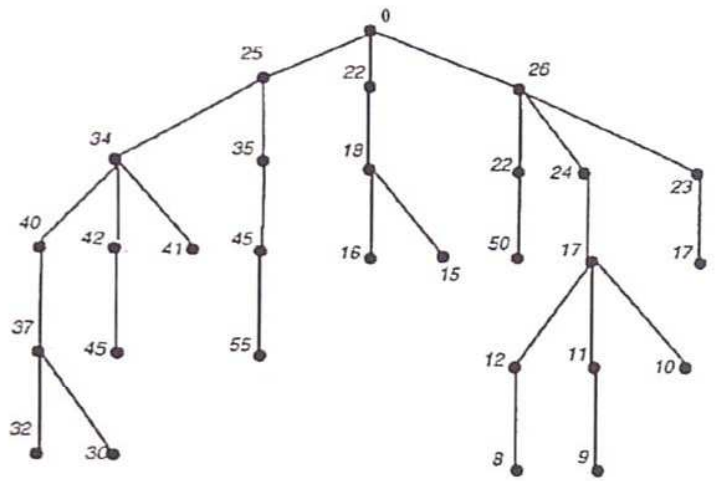
Plan

Recherche en profondeur et retour-arrière

- Implantation plus intelligente de l'algorithme du British Museum.
- Egalement appelé "algorithme par essais successifs", avec l'utilisation du "back-tracking".
- **Principe (parcours en profondeur)** :
  - A chaque échec, on réalise un retour au nœud précédent le nœud où l'on a constaté l'impasse et on choisit une autre règle de production parmi celles qui n'ont pas encore été utilisées ;
  - Cette nouvelle règle générera un autre nœud.
- Inconvénients :
  - Dans certains cas défavorables, on peut parcourir l'intégralité de l'espace d'état avant de trouver la solution ;
  - Algorithme n'utilisant pas la structure du problème pour éviter certains chemins absurdes ;
  - Algorithme garantissant d'aboutir à la solution uniquement si l'on effectue des tests d'occurence pour éviter d'éventuels bouclages.

Recherche en profondeur et retour arrière

Exemple : algorithme pour trouver une position dont la valuation est supérieure à 50 :



Recherche par essais successifs et problèmes NP

- La méthode des essais successifs s'intéresse à une mise en oeuvre de solution pour les problèmes NP : Non-Déterministes Polynomiaux.
- Exemples de problèmes NP : parcours dans un labyrinthe contenu dans une grille $n \times n$, les reines placées sur un échiquier $n \times n$, le voyageur de commerce devant visiter n villes, etc.

Mise en oeuvre d'un algorithme non déterministe

Soit un problème de taille n.
Soit $X = [x_1 \cdots x_m]$ avec $m = O(n^k)$ la forme de toute solution.
Soit P la propriété que doit vérifier X pour être solution.

```
procedure nonDet(i)
    y = UnChoix(ensemble des valeurs possibles de X[i]);
    si P est partiellement verifiee par [X[1] ... X[i-1] y]
        alors
        X[i] :=y;
        si i=m alors
          SUCCES;
        sinon
          nonDet(i+1);
      sinon
    ECHEC;
fin
```


Mise en oeuvre d'un algorithme non déterministe

1. Cet algorithme est non déterministe :
   - "UnChoix" délivre un élément de l'ensemble des valeurs possibles pour X[i].
   - "UnChoix" garantit qu'il existe des solutions ; l'exécution se termine par SUCCES.
   - "UnChoix" ne garantit pas que 2 exécutions consécutives construiront la même solution.
2. La complexité de cet algorithme est polynomiale.
   Si "UnChoix" est de coût $O(1)$, l'algorithme est de coût $O(m) = O(n^k))$

Mise en oeuvre séquentielle du non-déterminisme

Modèle général : Recherche des solutions dans un ensemble S.

A l'étape i, on a l'algorithme général suivant :

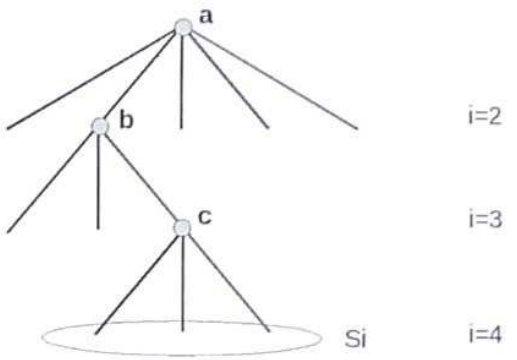
```
procedure chercher(int i)
    y : elements de Si
debut
    pour y parcourant Si faire
      si PP(X,i,y) alors
        X[i] :=y;
        modifier(y);
        si candidat(i) alors
          traiter_solution(i);
        sinon
          chercher(i+1);
        restituer(y);
fin
```


Mise en oeuvre séquentielle du non-déterminisme (2)

Notations du modèle général :

- PP(X,i,y) : vrai si (X[1] ... X[i-1],y) vérifie le prédicat partiel
- modifier(y) : prise en compte sur les variables globales du choix X[i]=y
- restituer(y) : annulation des modifications dues au choix X[i]=y
- candidat(i) : vrai si (X[1] ... X[i-1],X[i]) est solution ; candidat(i) est souvent de la forme i=n

Modèle général



Modèle général : arrêt sur la première solution

Si on veut s'arrêter dès qu'on a trouvé une solution acceptable :

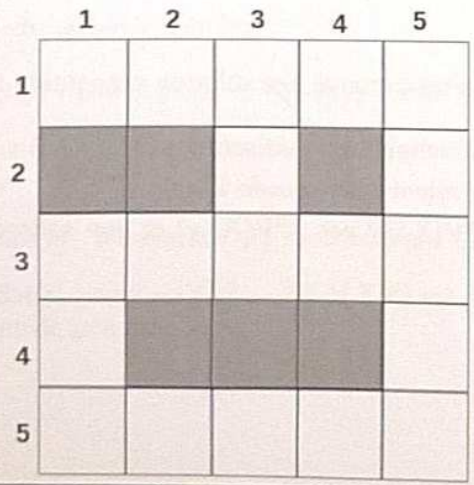
- On initialise à faux la variable globale *soltrouvée* ;
- *traiter_solution(i)* positionne *soltrouvée* à vrai ;
- on remplace le test PP(X,i,y) par : PP(X,i,y) et non *soltrouvée*.

Modèle général : recherche d'une solution optimale

- On gère une variable globale OPT qui contient la meilleure solution calculée depuis de le début de la recherche par essais successifs ;
- On remplace le test PP(X,i,y) par : PP(X,i,y) et *meilleur(X,i,y,OPT)* ; la condition *meilleur(X,i,y,OPT)* vaut faux si (X[1] ... X[i-1],y) n'est préfixe d'aucune solution meilleure que celle qui est représentée dans OPT ;
- *traiter_solution(i)* met à jour la variable globale OPT à l'aide de (X[1] ... X[i-1],X[i]).

Premier exemple : problème du labyrinthe

- Soit l'exemple suivant :



Problème du labyrinthe

- On veut **dénombrer** les chemins menant de la case (1,1) à la case (n,n) ne passant pas deux fois par la même case et utilisant les déplacements $\rightarrow$, $\uparrow$, $\leftarrow$, $\downarrow$.
- Les données :
  - $L[1 \cdots n, 1 \cdots n]$
  - $L[i,j] = \begin{cases} \text{vrai} & \text{case autorisée} \\ \text{faux} & \text{case interdite} \end{cases}$
  - Pour simplifier, on étend le labyrinthe en mettant un mur tout autour :

$$\begin{array}{lllll} \forall i,j & L[0,j] & = & \text{faux} \\ & L[i,0] & = & \text{faux} \\ & L[i,n+1] & = & \text{faux} \\ & L[n+1,j] & = & \text{faux} \end{array}$$

Labyrinthe : arbre de recherche par essais successifs

Labyrinthe : algorithme par essais successifs

Procédure de dénombrement par essais successifs :

- On considère le tableau $X[1..n^2]$ qui contient les cases successives du chemin courant.
- On initialise les variables suivantes :
  - nbrechemin :=0
  - X[1] :=(1,1)
- On commence avec : chercher(2)

Labyrinthe : algorithme par essais successifs

```
procedure chercher(int i)
    entier j, case y;
debut
    pour j:=1 jqa 4 faire // parcours des 4 voisins
      y:= Voisins(X[i-1],j);
      // fonction qui donne le jeme voisin d'une case (u,v)
      // ordre : droite bas gauche haut
      si Libre[y] et y not in X[1..i-2] alors
        X[i] :=y;
        si y=(n,n) alors
          nbrechemin +=1;
        sinon
          chercher(i+1);
fin
```


Labyrinthe : complexité

- Complexité : 4^{n^2}
  (correspond à la hauteur de l'arbre)
- Si pas de case noire, alors le chemin a une longueur de n^2.
- Remarque : X est gérée avec une pile

Problème du voyageur de commerce

- **Problème** : soit n villes, un représentant de commerce doit, en partant de la ville v, effectuer une tournée en passant une fois et une seule par toutes les villes et revenir en v en minimisant la distance parcourue.
- Modélisation par un graphe (non orienté) :
- Il faut trouver un cycle hamiltonien (circuit qui passe une fois et une seule par chaque sommet du graphe)

Problème du voyageur de commerce

Les différentes étapes :

1. Dessiner l'arbre de recherche
   - On ne fait pas figurer les nœuds qui contredisent le fait que l'on cherche un circuit hamiltonien.
   - La fonction d'évaluation est monotone croissante : on ne fait pas figurer les nœuds ne permettant pas d'obtenir mieux que le minimum déjà trouvé.
2. Ecrire l'algorithme

Problème du voyageur de commerce : arbre de recherche

Problème du voyageur de commerce : algorithme

```
procedure tournee(int i)
    y : elements de Si
debut
    pour y appartenant Voisins(X[i-1]) faire
     Si = Voisins(X[i-1])
      si  (y pas dans X[0...i-1] ou (i=n et y=1))
          et lg + v(X[i-1],y) < lgopt  PP(X,i,y)  alors
          X[i] :=y;
          lg := lg + v(X[i-1],y); modifier(y)
          si i=n alors  candidat(i)
            lgopt := lg;  traiter_solution(i)
            XOPT := X;
          sinon
            tournee (i+1); chercher(i+1)
          lg := lg -v(X[i-1],y); restituer(y)
fin
```

- X : tableau d'entiers de 0 à n, X[0] :=1 ;
- $lgopt = \infty$
- $lg := 0$
- tournee(1)
- XOPT contient la tournée optimale

Problème du voyageur de commerce : complexité

Complexité :

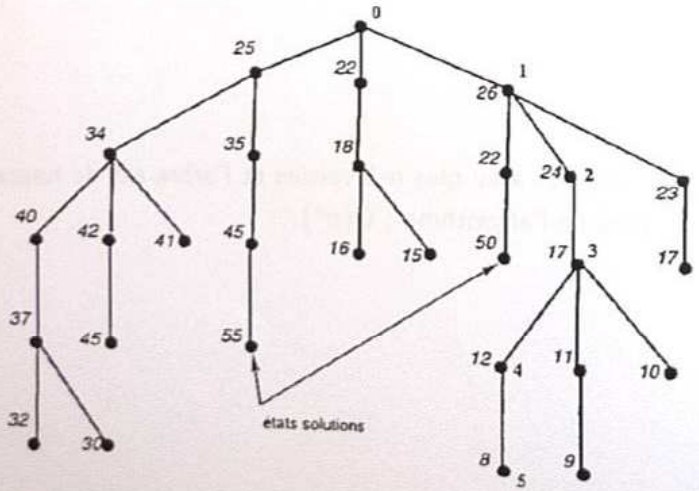
- Chaque sommet a au plus n-1 voisins et l'arbre est de hauteur n
- Complexité de l'algorithme : $O(n^n)$

Remarque : recherche en escalade

- **Principe** : méthode de recherche en profondeur dans laquelle on introduit une fonction heuristique pour choisir à chaque étape le nœud à générer au lieu de générer un nœud en prenant une règle de production au hasard.
- Problème du voyageur de commerce avec recherche en profondeur et retour en arrière : génération potentielle de n ! nœuds.
- Si on prend comme heuristique de générer à chaque fois le nœud qui ajoute à la liste des villes déjà visitées la ville la plus proche de la dernière traversée, l'algorithme a une complexité en $O(n^2)$.

Recherche en escalade

Exemple : algorithme pour trouver une position dont la valuation est supérieure à 50 :



Inconvénient de la méthode par essais sucsessifs

- Le principal inconvénient de la méthode par essais successifs : son coût.
  On constate en dessinant l'arbre de recherche que le coût de ces algorithmes est d'ordre exponentiel.
- En remarquant que la recherche d'une solution optimale pour le problème du voyageur de commerce a permis d'éviter de parcourir tout l'arbre, on peut envisager d'utiliser la notion d'heuristique afin, dans certains cas, d'élaguer l'arbre de recherche : cf. prochaine partie du cours.

Problème des 8 reines (1)

Enoncé du problème :

- Il faut placer 8 reines dans un échiquier (matrice 8x8) sans qu'aucune d'entre elles ne soient sur une même ligne, une même colonne et une même diagonale.
- Ainsi : une reine placée dans la case d'indice (i,j) condamne la ligne i, la colonne j et les diagonales passant par la case (i,j).
- La diagonale positive est celle qui forme un angle de 45 ° et la diagonale négative celle qui forme un angle de -45 °.

Problème des 8 reines : arbre de recherche

- Les états formant l'espace de recherche sont les différents échiquiers avec un nombre de reines déjà placées entre 0 et 8. L'espace de recherche est très grand mais fini.
- L'état initial représente un échiquier vide. Les états solutions sont ceux représentant les 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.
- Si on génère toutes les solutions possibles (C_{64}^{8}), il y a 4 426 165 368 possibilités !

Problème des 8 reines : première amélioration

- On peut améliorer l'algorithme en remarquant que chaque ligne d'une solution contient exactement une seule reine. Ceci permet de réduire la représentation de l'échiquier à un simple vecteur de 8 éléments où les indices désignent les lignes et les contenus les colonnes des reines déjà placées. Par exemple : $\{3, 1, 6, 2, 8, 6, 4, 7\}$ (solution incorrecte)
- Questions :
  1. Ecrire l'algorithme.
  2. Complexité ?

Problème des 8 reines : deuxième amélioration

- Après cette première amélioration, on peut encore restreindre le nombre de possibilités en se restreignant aux permutations de 8 nombres (pas 2 reines sur la même colonne).
- Il reste encore un défaut majeur : la solution n'est vérifiée qu'une fois que les 8 reines sont placées alors que dès le placement des 2 premières reines, on peut constater qu'on ne peut pas aboutir à une bonne solution (on traite alors $6! = 720$ cas inutilement).

Problème des 8 reines : troisième amélioration

- On peut remarquer que :
  - pour une diagonale négative, la différence des indices est constante ;
  - pour une diagonale positive, la somme des indices est constante.

  On utilise cette propriété pour écrire l'algorithme par essais successifs

Problème des 8 reines : algorithme de placement

Soit $X[1 \cdots i-1]$ tel que $X[i]$ est la position de la reine dans la ième ligne.

L'algorithme de recherche pour savoir si la position convient pour la ligne i sachant les choix déjà faits en $1 \cdots i-1$ est :

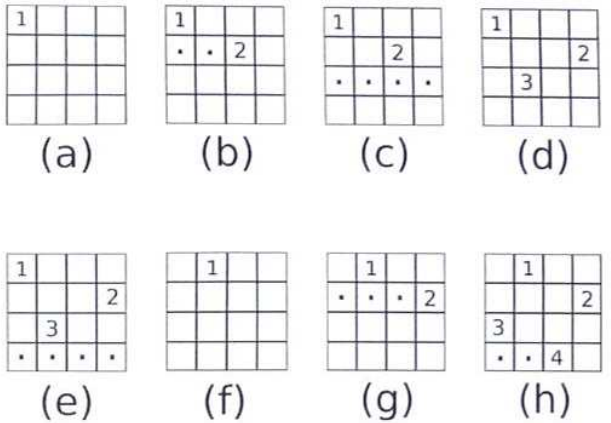
```
fonction place(i,y)
  entier k := 1;
  boolean trouvePlace := vrai:
  tant que k<i faire :
      si X[k] = y // meme colonne
        ou abs(y-X[k]) = abs(i-k) // meme diagonale
      alors
        trouvePlace:= faux;
      k:=k+1;
   retourner trouvePlace;
```


Problème des 8 reines : algorithme général

```
procedure nreine(i)
    entier k;
    pour y de 1 a n faire
      si place(i,y) alors
        X[i]:=y;
        si i = n alors
          ecrire(X);
        sinon
          nreine(i+1);
```


Problème des 8 reines : exemple

Solution de backtracking avec 4 reines :



Problème des 8 reines : exemple

Problème des cruches d'eau (1)

Enoncé du problème :

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

Problème des cruches d'eau (2)

Les manipulations permises sont données par les règles suivantes (Q(c) représente la quantité d'eau courante dans la cruche c) :

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 $Q(B) < (4 - Q(A))$;

Problème des cruches d'eau : parcours de l'arbre

- Un état de l'espace de recherche peut être représenté par un couple (x,y) où x est la quantité d'eau contenue dans la cruche A et y la quantité d'eau dans la cruche B.
- L'état initial est donc (0,0) et les états solution sont de la forme (2,n) avec n compris entre 0 et 3.
- La technique du backtracking consiste à appliquer les règles 1 à 8 (dans cet ordre par exemple), pour chaque nouvel état visité.

Problème des cruches d'eau : espace de recherche

Problème des cruches d'eau : complexité

Exercice : calculer la complexité de l'algorithme.

Plan

Arbres de jeux

- On considère les jeux de stratégie entre **2 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 nœud 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** (joueur A dans la suite du cours), l'autre joueur (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 l'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 choisir parmi les fils de la configuration courante celui qui porte la plus grande valeur, et inversement si c'est un joueur minimsant.

Algorithme du Min-Max : attribution des valeurs aux différentes configurations

- On commence par attribuer des valeurs aux feuilles (+1 si A gagne, -1 si A perd et 0 si nul). Les règles du jeu permettent de déterminer si une configuration feuille représente une partie gagnante ou perdante.
- Les valeurs sont propagées vers les nœuds ascendants jusqu'à la racine de la façon suivante :
  - si c'est au tour de A de jouer, le nœud correspondant prend la plus grande des valeurs de ses fils ;
  - si c'est au tour de B de jouer, le nœud correspondant prend la plus petite des valeurs de ses fils.

Algorithme du Min-Max : partie gagnante/perdante

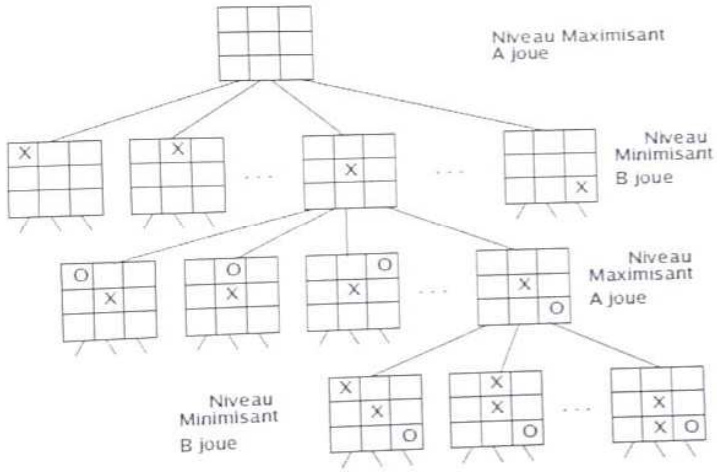
- Les différentes configurations :
  - **Si la racine prend la valeur -1**, le joueur A est assuré de perdre si B ne fait pas d'erreurs. Dans ce cas, c'est B qui a une stratégie gagnante.
  - **Si la racine prend la valeur 0**, aucun des 2 joueurs n'a de stratégie gagnante, mais tous deux peuvent s'assurer, au pire, d'un match nul en jouant aussi bien que possible (cf. cas du jeu Tic-Tac-Toe).
- Remarque : pour propager les valeurs de bas en haut, l'algorithme Min-Max parcourt l'arbre des configurations avec un parcours **post-ordre** : avant d'évaluer un nœud donné, il faut d'abord évaluer tous ses fils.

Algorithme min-max : exemple du jeu Tic-Tac-Toe

- **Principe :** Le jeu Tic-Tac-Toe (ou morpion) consiste à placer des croix et des cercles sur une grille de 9 cases (3x3) jusqu'à ce que l'un des joueurs aligne trois de ses symboles.
- **Espace de recherche :**
  - 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).

Tic-Tac-Toe : arbre de recherche

Illustration des premiers niveaux de l'arbre de recherche :



Tic-Tac-Toe : application de l'algorithme Min-Max

- Supposons qu'on soit arrivé à la configuration suivante et que ce soit au tour de A de jouer :

X		X
	X	O
O		O

- Pour évaluer cette configuration, il faut parcourir en post-ordre l'arbre ayant comme racine cette configuration et propager les valeurs des feuilles vers les nœuds ascendants (cf. slide suivante) ;

Tic-Tac-Toe : application de l'algorithme Min-Max

Algorithme Min-Max : exemple du jeu de Nim

Principe :

- 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 (constante inférieur à n, fixée au début du jeu).
- Celui qui peut ramasser les dernières billes de la table remporte la partie.

Jeu de Nim : espace de recherche

- Un état de l'espace de recherche peut être représenté uniquement par le nombre de billes présentes sur la table à un instant donné.
- Soit nb le nombre de billes sur la table. A l'état initial, on a nb=n. Un état solution est représenté par nb=0 ;
- On prend comme convention que A est un joueur maximisant et B un joueur minimisant. Supposons que A commence la partie.

Jeu de Nim : arbre de recherche

Arbre de recherche pour n=4 et k=3 :

Algorithme du Min-Max

Déroulement de l'algorithme :

- **Si l'arbre de recherche n'est pas trop grand** : on peut lancer une seule fois la fonction MinMax pour évaluer tous les nœuds possibles et stocker cette information en mémoire. Lors du déroulement du jeu, il suffit de choisir parmi les configurations courantes celles qui portent la plus grande/petite valeur selon le joueur.
- **Si l'arbre de recherche est trop pour être parcouru entièrement** (souvent le cas, par exemple le jeu d'échecs) : la fonction MinMax doit être appelée au cours du jeu à chaque fois que la machine doit jouer. On doit alors limiter le parcours à une profondeur maximale, en fonction des capacités de l'ordinateur et le temps de réponse voulu.

Algorithme du Min-Max

```
type typeMode = {Min,Max};

Fonction MinMax(J : jeu; mode : typeMode; Niveau : entier)

  val : reel;

  si J est une feuille
    retourner (Cout(J));
  sinon
    si Niveau = 0
      retourner (Estimation(J));
    sinon
      si mode = Max
        val := -infini;
      sinon
        val := +infini;
      pour chaque fils K de J faire
        si mode = Max
            val := Max(val, MinMax(K,Min,Niveau-1));
        sinon
            val := Min(val, MinMax(K,Max,Niveau-1));
      retourner val;
```


Algorithme du Min-Max et fonctions d'estimation

Comment évaluer une configuration atteinte si ce n'est pas une feuille et qu'on est à la profondeur maximale ?

- On utilise des **fonctions d'estimation** qui s'appliquent à une configuration donnée et retournent une valeur estimée de sa qualité (proche de -1 pour une configuration favorable au joueur minimisant et proche de 1 pour une configuration favorable au joueur maximisant).
- Ces fonctions d'estimation dépendent de chaque jeu et influent de façon importante la qualité de jeu de la machine (ex : jeu d'échecs).

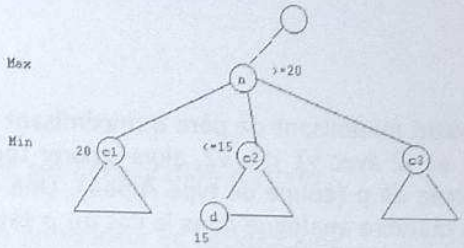
Algorithme du Min-Max

- La fonction MinMax retourne le coût de la configuration J.
- **Appel initial** : on donne la configuration J que l'on veut évaluer, le joueur (maximisant ou minimisant) qui va jouer et la profondeur maximale de l'arbre que l'on veut atteindre (Niveau).
  - La fonction Cout(J) est appliquée uniquement à une feuille et retourne -1, 1 ou 0.
  - La fonction Estimation(J) s'applique à n'importe quelle configuration et retourne une valeur comprise entre -1 et 1.

Elagage alpha-beta

- Le principe de l'élagage Alpha-Bêta (ou coupe de branche "pruning") consiste à éliminer une grande partie de l'arbre de recherche dans l'algorithme MinMax en ignorant certains descendants d'un nœud.

Elagage alpha-beta : illustration



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 nœud maximisant) ;
3. En explorant c2 et ses fils, on tombe sur un nœud d de valeur 15. Comme d est un fils de c2, on aura c2 $<=$ 15 (nœud minimisant) ;
4. A ce niveau, on peut abandonner l'exploration des autres fils de c2 et passer directement à celle de c3 car la valeur de c2 ne peut pas influencer celle de n.

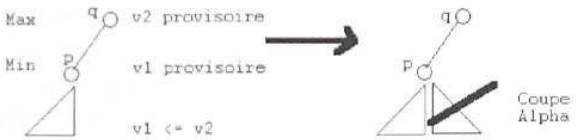
Elagage alpha-beta : règles de calcul

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

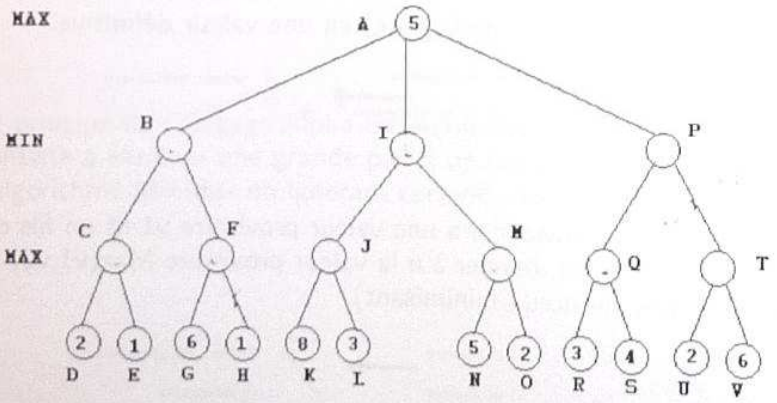
2. Si un nœud maximisant n a une valeur provisoire v1 et un fils de valeur définitive v2, donner à n la valeur provisoire Max(v1,v2) (même principe avec un nœud minimisant).

Elagage alpha-beta : règles de calcul (2)

1. Si p est un nœud minimisant de père q maximisant avec des valeurs provisoires v1 et v2 avec v1 <= v2, alors ignorer toute la descendance encore inexplorée de p (coupe de type Alpha). Une coup de type Bêta est définie de manière analogue dans le cas où p est maximisant et q minimisant et v1 >= v2.



Elagage alpha-beta : exemple



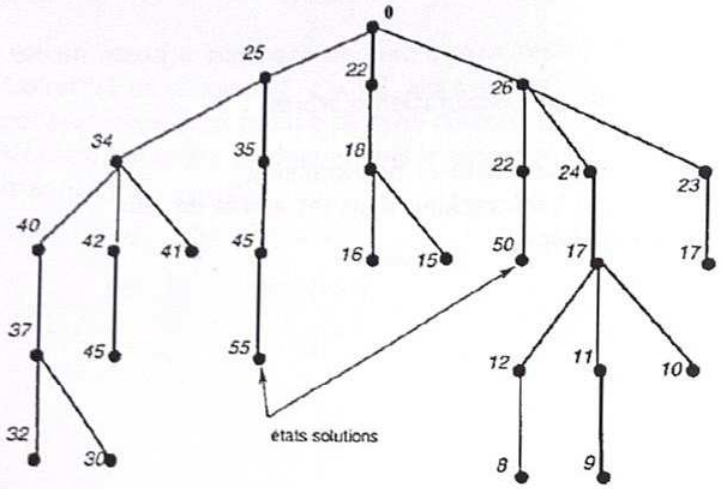
Plan

Recherche en largeur

- **Principe** : générer l'ensemble des nœuds niveau par niveau.
- Avantages :
  - L'algorithme garantit de trouver une solution même s'il existe un cycle ;
  - Solution la moins profonde.
- Inconvénients :
  - Grande consommation mémoire ;
  - Algorithme pouvant conduire au parcours de l'intégralité de l'arbre dans les cas défavorables ;
  - Alternative : solution mixte.

Recherche en largeur

Exemple : algorithme pour trouver une position dont la valuation est supérieure à 50.



Recherche en largeur : exercice

- **Problème** : on voudrait passer du nombre 15 au nombre 4 en utilisant uniquement les fonctions f et g suivantes : $f(x) = 3x$ et $g(x) = xdiv2$.
- 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 fonctions f et g menant vers 2 autres états.
- L'espace de recherche est un graphe infini, donc c'est la recherche en largeur qui est recommandée. De plus, on trouvera également le nombre minimal d'application des fonctions f et g pour passer de 15 à 4.

Recherche en largeur : exercice (2)

- On prend comme convention que les arcs sortant à gauche représentent les applications de la fonction f et ceux sortant à droite représentent les applications de g :