Chapitre IV
Recherches Guidées par Heuristiques

IV.1. Notion d'heuristique

Une heuristique est une technique qui améliore l'efficacité d'un processus de recherche, en sacrifiant éventuellement l'exactitude ou l'optimalité de la solution.

Pour des problèmes d'optimisation (NP-complets) où la recherche d'une solution exacte (optimale) est difficile (coût exponentiel), on peut se contenter d'une solution satisfaisante donnée par une heuristique avec un coût plus faible.

Certaines heuristiques sont polyvalentes (elles donnent d'assez bons résultats pour une large gamme de problèmes) alors que d'autres sont spécifiques à un type particulier de problème.

Dans le chapitre précédent, l'utilisation de fonctions d'estimation dans Minmax peut être vue comme étant une technique heuristique pour évaluer plus rapidement une configuration donnée dans un jeu de stratégie.

Les heuristiques peuvent aussi être utilisées dans certains algorithmes de recherche exacts, pour accélérer la découverte de solutions optimales, ce qui semble paradoxal (voir Algorithme A* plus loin).

IV.1.1. Algorithmes voraces (gloutons)

C'est une méthode heuristique simple et d'usage général (appelée aussi algorithme du plus proche voisin). Son principe est qu'à chaque étape durant le processus de recherche, l'option localement optimale est choisie. Ce principe est répété jusqu'à trouver une solution (exacte ou non). Les choix faits durant le processus de recherche ne sont jamais remis en cause (pas de retour arrière).

Plusieurs algorithmes importants sont issus de cette technique :
– Algorithme de Dijkstra pour les plus courts chemins entre un sommet et tous les autres sommets d'un graphe.
– Algorithme de Kruskal pour les arbres sous-tendants minimaux: dans un graphe orienté, trouver l'arbre de poids minimal connectant tous les sommets

Exemple 1:
On veut totaliser une somme d'argent S en utilisant un nombre minimal de pièces appartenant à un ensemble donné.
A chaque étape on choisit la plus grande pièce de valeur inférieure ou égale à S tout en mettant à jour la nouvelle somme à atteindre (S – la valeur de la pièce choisie):

Pour S=257 DA et pièces={100DA, 50DA, 20DA, 10DA, 5DA, 2DA, 1DA}
à l'étape 1 : S1 = 257 , on choisit 1 pièce de 100DA
à l'étape 2 : S2 = 157 , on choisit 1 pièce de 100DA
à l'étape 3 : S3 = 57 , on choisit 1 pièce de 50DA
à l'étape 4 : S4 = 7 , on choisit 1 pièce de 5DA
à l'étape 5 : S5 = 2 , on choisit 1 pièce de 2DA
La solution trouvée est donc 2x100 DA , 1x50 DA , 1x5 DA et 1x2 DA

Exemple 2:
Voyons comment un simple algorithme vorace, dérivé de celui de Kruskal, trouve généralement une solution assez bonne au Problème du Voyageur de Commerce (PVC).
Le PVC peut s'énoncer comme suit:

> étant donné un ensemble de villes avec des distances données entre chaque couple de villes, trouver la plus petite tournée passant par toutes les villes. Cela revient à trouver le plus petit cycle simple passant par tous les sommets d'un graphe complet (un graphe où il existe une arête entre chaque couple de sommets)

Le principe est de commencer avec un ensemble d'arêtes vide V et de l'enrichir à chaque étape par l'arête de poids minimal et non encore considérée, vérifiant les deux conditions suivantes:
a) elle ne doit pas formée un cycle avec les arêtes déjà choisies (dans V) sauf si le cycle construit passe par tous les sommet, auquel cas c'est le résultat final,
b) elle ne doit pas être la 3e arête choisie incidente à un même sommet (car dans un cycle, tous les sommets doivent avoir un degré = 2).

Par exemple, considérons le graphe complet dont la matrice (symétrique) des coûts est la suivante:

	a	b	c	d	e	f
a		3	10	11	7	25
b			6	12	8	26
c				9	4	20
d					5	15
e						18
f						

Les arêtes seront choisies dans l'ordre croissant des coûts associés: {ab,ce,de,bc,df,af}
Donnant ainsi le cycle résultat: { a b c e d f a } de longueur 58.
Le cycle optimal était: { a b c f d e a } de longueur 56.

Dans le reste de ce chapitre, nous allons présenter l'utilisation d'heuristiques dans les algorithmes d'exploration de graphes. Nous appellerons ces méthodes des recherches guidées.

IV.2. Les Recherches Guidées

Il y a deux grandes classes de recherches guidées: celles utilisant les heuristiques pour

accélérer la recherche de solutions pas forcément optimales (les parcours en profondeur et en largeur avec fonctions d'estimations trouvent rapidement un état solution) et celles utilisant les heuristiques pour accélérer la recherche de solutions optimales (Branch & Bound et A* trouvent les plus courts chemins entre l'état initial et un état solution).

IV.2.1. DFS avec estimation (Hill-Climbing)

C'est une recherche en profondeur avec fonction d'estimation pour ordonner les alternatives à chaque étape. On l'appelle aussi « Hill-Climbing ».

La fonction d'estimation (qui doit être donnée en fonction du problème) sert à estimer la proximité d'un sommet solution à partir du sommet courant dans l'espace de recherche. Ainsi à partir d'un sommet x, la procédure favorise en premier lieu, l'exploration des successeurs de x ayant la plus petite estimation.

Par exemple, pour le problème du labyrinthe, la distance euclidienne peut être utilisée comme fonction d'estimation de proximité de la case solution. A chaque alternative dans le parcours du labyrinthe, on commence d'abord par celle qui est la plus proche (selon la distance euclidienne) de la case solution.

L'algorithme suivant permet d'effectuer une recherche en profondeur avec estimation, pour rechercher un sommet solution (BUT). L'algorithme utilise une pile et une fonction d'estimation f(x) pour ordonner localement les choix:

```
CreerPile(P);
Empiler(P, Racine);
Trouv ← FAUX;
TantQue ( Non PileVide(P) et Non Trouv )
  Depiler(P,x);
  Si ( x <> BUT )
        Si état[x] = « non visité »
            état[x] ← « visité »;
            Soient y1,y2,...yk les successeurs de x non encore vistiés
            Empiler les yi en ordre décroissant de leurs estimations.
        Fsi
  Sinon
      Trouv ← VRAI
  Fsi
FTQ
```

IV.2.2. BFS avec estimation (Best First Search)

C'est une recherche en largeur avec fonction d'estimation. On l'appelle aussi « Best

First Search ».

A chaque fois qu'on génère les successeurs d'un sommet durant l'exploration de l'espace de recherche, on les rajoute à une file de priorité contenant tous les sommets générés mais pas encore explorés. La file est ordonnée par la fonction d'estimation, de sorte que le prochain sommet à visiter soit toujours le « meilleur » parmi tous ceux déjà générés dans la file.

L'algorithme suivant permet de réaliser une recherche de type « Best First Search » pour rechercher un sommet solution (BUT). La seule différence avec la recherche en largeur pure, est le fait d'utiliser une file de priorité ordonnée par les valeurs de la fonction d'estimation f donnée.

```
CreerFile(F)          /* F une file de priorité : ces éléments sont des
                         couples de la forme <Sommet,Estimation>  */
Enfiler( F , <Racine,f(Racine)> );
Trouv ← FAUX;

TantQue ( Non FileVide(F) et Non Trouv )
  Defiler( x );       /* x est le sommet avec la plus petite estimation */
  Si ( x <> BUT )
       Si ( état[x] = « Non visité » )
            état[x] ← « Visité »;
            Soient y1,...yk les succ de x non encore visités;
            Pour i=1,k
                 Enfiler( F , <yi,f(yi)> )  /* f : fct d'estimation */
            Fp
       Fsi
  Sinon
      Trouv ← VRAI
  Fsi
FTQ
```

IV.2.3. La famille de méthodes "Branch & Bound / A*"

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

L'algorithme de base est le suivant :

```
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. A 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.

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)

où g(x) représente le coût du chemin entre la racine est x et
h*(x) représente une estimation du coût restant entre x et un éventuel sommet solution (BUT) accessible à partir de x.

Ainsi f(x) serait une estimation du coût d'un chemin entre la Racine et le BUT passant par le sommet 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 sous-estimation.
C'est-à-dire si h*(x) = v alors le coût réel du trajet entre x et le BUT est forcément ≥ v.

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 sous-estimation.

Comme le problème du plus court chemin vérifie le principe d'optimalité (dans un chemin optimal, tout sous-chemin doit aussi être optimal), on peut réorganiser 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 finalement 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.

etiq: 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).

La méthode A* est tout simplement « Branch and Bound » avec sous-estimation et programmation dynamique.

Autre version de l'algorithme A*.

On maintient deux listes Ouvert (O) et Fermé (F) dont l'une est une file d'attente avec priorité (Ouvert).

g représente le coût réel du trajet effectué entre la racine et le nœud n .
h représente l'estimation de la distance restante entre n et un nœud solution.
f = g+h représente la fonction d'estimation pour le choix du prochain nœud.

1. O ← {le nœud racine s}.
2. Retirer un nœud n de O qui a la plus petite valeur de f(n).
Si (n = but) stop avec succès
Sinon
- Insérer n dans F
- Générer les successeurs m de n
- Pour chaque successeur m :
Si (m n'est pas dans O et m n'est pas dans F)
. Insérer m dans O
. Attacher un pointeur arrière vers n
Fsi
Si (m est dans O OU m est dans F)
. Changer le chaînage arrière selon le coût de la plus petite branche (g)
Fsi
Si (m est dans F et son chaînage arrière est modifié)
. l'enlever de F
. Enlever tous les nœuds de la descendance de m qui sont dans O et dans F.
. Insérer m dans O.
Fsi
- FP
Fsi
3. Aller à 2 .

Remarques à propos de g :
La fonction g nous laisse choisir quel nœud élargir sur la base de tout le chemin depuis la racine.
Si nous voulons arriver rapidement à une solution (pas forcément optimale), il suffit de donner à g la valeur 0 (Cas de l'algorithme Best First Search) .
Si nous voulons trouver un chemin (solution) avec le plus petit nombre de pas possible, il suffit de donner à g la valeur 1.
Si nous voulons trouver le chemin le moins coûteux, il suffit de donner à g le coût du trajet (ou de l'opération).

Remarques à propos de h :
Si h' est parfait, A* converge immédiatement vers la solution sans retour arrière. Meilleur est h' plus rapidement nous trouverons la solution.
Si h' vaut 0, la recherche sera contrôlée uniquement par g. Si g vaut 0, la recherche est aléatoire. Si g vaut toujours 1, la recherche se fera en largeur.
** Si h' ne surestime jamais h (distance réel), A* est assuré de trouver un chemin optimal vers un but s'il existe (propriété d'admissibilité) .

Exemples:

1. Le labyrinthe:
   Trouver le plus court chemin entre une case de départ (1,1) et une case d'arriver (5,4) dans le labyrinthe représenté par une grille de 5x5. Les cases fermées sont (2,2) (3,2) (5,2) (5,3) (1,4) (3,4) et (4,4). Les déplacements possibles sont dans la verticale et dans l'horizontale.
   La fonction g(x) = le nombre de cases parcourus dans ce chemin depuis la racine jusqu'à x. La fonction h*(x) = la distance euclidienne entre la case x et la case (5,4).

2. Le taquin (en prenant par exemple h*(x) = le nombre de pièces mal placées)

3. Le Problème du Voyageur de Commerce (PVC)