# INFO Camel MP\*

1. Retour sur trace

Arbre des configurations

Le **retour sur trace** (backtracking) est une technique de programmation pertinente lorsqu’on cherche la solution d’un problème parmi un grand nombre de possibilités pouvant être représentées par un arbre des configurations possibles. Les feuilles correspondent aux configurations complètes, et les nœuds internes aux configurations partielles.

**Exemple** : dans le problème des 8 dames, on cherche comment positionner 8 dames sur un échiquier sans qu’aucune ne se menace. On définit alors une configuration comme la liste des lignes des k dames présentes sur les k premières colonnes, dans l’ordre des colonnes. Une configuration est complète si k = 8.

Parcours de l’arbre

Enumérer et tester par force brute toutes les configurations complètes peut être très long : il y en a 88 dans le problème des 8 dames. L’idée du retour sur trace est de parcourir l’arbre des configurations en profondeur. Dès qu’une configuration ne respecte pas les contraintes du problème le parcours saute l’exploration de toute la partie de l’arbre en dessous, ce qui permet d’éliminer un nombre potentiellement grand de configurations complètes.

En pratique on ne construit pas explicitement l’arbre des configurations, et ce parcours est une description abstraite da la structure récursive du programme qui va implémenter ce retour sur trace.

Exercice :

1. Ecrire une fonction prenant en argument deux entier n et m et renvoyant la liste de toutes les listes de n entiers distincts entre 0 et m-1
2. Ecrire une fonction n\_dames prenant en argument un entier n et renvoyant la liste des configurations complètes valides du problème des n dames.
3. Calculer le nombre de configurations considérées lors d’un appel à n\_dames.
4. Ecrire une variante de la fonction précédente renvoyant une seul configuration complète valide (s’il en existe une)
5. Ecrire des variantes des fonctions précédentes pour lesquelles on ajoute la contrainte que les dames ne peuvent pas ce menacer comme des cavaliers.

Tas binaire

Def :

Un **tas** **binaire** est un arbre binaire vérifiant les propriétés suivantes :

1. A chaque nœud est associé une clé
2. La clé de chaque nœud est supérieure à la clé de chacun de ses fils
3. L’arbre est **complet** **à** **gauche** : tous les niveaux de profondeur sont complets sauf éventuellement le dernier, rempli de gauche à droite.

**Remarques** : Les deux premières propriétés rendent pertinente l’implémentation d’une file de priorité par un tas. La troisième propriété rend pertinente l’implémentation d’un tas par un tableau.

Représentation d’un tas par un tableau

Il suffit d’attribuer à chaque nœud un indice. On choisit de numéroter les nœuds du niveau le plus haut au niveau le plus bas, et pour chaque niveau de la gauche vers la droite.

**Exercice** : Le tableau [| 20 ;17 ;8 ;12 ;15 ;3 ;2 ;1 ;4 ;13|] représente-t-il un tas ? si oui lequel ?

**Exercice** : Pour un nœud d’indice i, donner (s’il existe), l’indice de son fils gauche, de son fils droit, de son père.

G : i -> 2i + 1

D : i -> 2i + 2

P : i -> (i-1) //2

**Remontée**

Def :

Un **i-sur-tas** est un arbre qu’on peut transformer un tas en remplaçant le nœud i par une valeur inferieur. Autrement dit, c’est un arbre vérifiant les propriétés d’un tas, sauf éventuellement au niveau des inégalités entre le nœud i et ses ascendants.

Lorsqu’on modifie un tas, on peut se retrouver avec un i-sur-tas. On peut alors obtenir un tas avec l’opération **remonter**, qui permute le nœud problématique et son père jusqu’à obtenir un tas ;

**Exercice** : Ecrire remonter, qui prend en argument un i-sur-tas sous forme de tableau et l’entier i, et qui permute les éléments du tableau de façon à en faire un tas.

Entassement

Def :

Un i-sous-tas est un arbre qu’on peut transformer en tas en remplaçant le nœud i par une valeur supérieur. Autrement dit, c’est un arbre vérifiant les propriétés d’un tas, sauf éventuellement au niveau des inégalités entre le nœud i et ses descendants.

Lorsqu’on modifie un tas, on peut se retrouver avec un i-sous-tas. On peut alors obtenir un tas avec l’opération **entasser**, qui permute récursivement le nœud problématique et son fils maximum jusqu’à obtenir un tas ;

Tri par tas

Pour trier un tableau t :

* On initialise une variable taille à la taille du tableau ;
* On entasse, de la droite vers la gauche, chaque élément du tableau, de manière à obtenir un tas ;
* On met l’élément d’indice 0, maximal, à sa place, en le permutant avec le dernier élément. On décrémente la taille pour s’assurer que l’élément maximal ne soit plus compris dans le tas, donc plus déplacé. On entasse l’élément arrivé en position 0 pour retrouver une structure de tas ;
* On itère l’étape précédente pour placer tous les éléments à leur place.

Implémenter le tri par tas, et déterminer sa complexité.

Implémentation des files de priorité simples

type (‘a,’b) file\_prio = {mutable taille : Int ; tas : (‘a\*’b) option array}

ce type enregistrement a deux champs, taille et tas. Le mot-clé mutable permet de modifier le champ taille, en écrivant par exemple f.taille<-f.taille +1

Implémenter les Operations suivantes :

* Créer\_file\_prio(c) : renvoie une file de priorité vide de capaciter c

Graphe

Principe

Un **parcours de graphe** est une stratégie d’exploration des sommets d’un graphe. Les arrêtes ou des arcs empruntés lors du parcours forment une **arborescence**, ie un ensemble d’arbres, chaque arbre correspondant à l’exploration d’une partie du graphe depuis un somment initial.

Une telle arborescence peur être représentée efficacement par un tableau associant à chaque sommet son père. Les sommets initiaux sont dans cette représentation traités comme leur propre père.

On distingue deux familles principales de parcours :

* Le parcours en largeur, ou on explore tous les sommets à distances k du sommet initial s avant de passer à ceux de distance k+1.
* Le parcours en profondeur, ou on visite à chaque etape un successeur du dernier sommet visité qui en a un.

Arborescence en largeur

Parcours\_largeur(G)

Chaque sommet est non marque

Chaque sommet est son propre père

Soit F une file vide

Pour chaque sommet s de G

Si s est non marque

On marque s

On enfile s dans F

Tant que F est non vide

On defile un sommet u dans F

Pour chaque v successeur de u

Si v est non-marque

On marque v

Le père de v devient u

On enfile v dans F

Arborescence en profondeur recursif

Visiter(u)

On marque u

Pour chaque v successeur de u

Si v est non marqué

Le père de v devient u

Visiter (v)

Parcours\_profondeur(G)

Chaque sommet est non marqué

Chaque sommet est son propre père

Pour chaque sommet s de G

Si s est non marque

Visiter(s)

Renvoyer P

Arborescence en profondeur, pile

Parcours\_profondeur(G)

Chaque sommet est non marque

Chaque sommet est son propre père

Soit P une pile vide

Pour chaque sommet s de G

Si s est non marque

On empile s dans P

Tant que P est non vide

On depile un sommet u de P

Si u est non marque

On marque u

Pour chaque v successeur de u

Si v est non-marque

On empille v dans P

Le père de v devient u

Tri topologique

Un **tri** **topologique** sur un graphe orienté acyclique est un ordre total sur les sommets compatible avec les arcs : si (u,v) est un arc, on doit avoir u < v. Un tel ordre existe mais n’est pas unique.

Un tri topologique peut être obtenu à partir d’un parcours en profondeur récursifs : on traite chaque sommet **après** ses successeurs. Les sommets seront alors traités dans l’ordre inverse d’un tri topologique.

On peut par exemple traiter chaque élément en l’ajoutant au début d’une liste pour récupérer un tri topologique sous forme de liste.

Visiter(u)

On marque u

Pour chaque v successeur de u

Si v est non marqué

Le père de v devient u

Visiter (v)

On ajoute u en tête de L

Tri\_topologique(G)

Chaque sommet est non marqué

Soit l une liste vide

Pour chaque sommet s de G

Si s est non marque

Visiter(s)

(convention n sommet m arrête/arc)

Parcours en O(n+m)

Preuve de correction

Soit (u,v) un arc de G. on considère le moment ou cet arc est explorer, ie ou v apparait dans la liste d’adjacence de u lors de l’appel *visiter(u)*

* Si v est non marqué, alors visiter(v) est appelé et ajoute v à L, puis sera ajouté plus à gauche.
* Si v est marqué et est déjà dans L, de la même façon, u sera plus tard ajouté plus à gauche
* Si v est marqué et n’est pas dans L, alors u est un successeur de v, puisqu’entre le moment ou un sommet est marqué et le moment où il est ajouté à L, on ne visite que ses descendant. On en déduit que G contient un cycle, ce qui contredit l’hypothèse d’acyclicité.

Def :

Un graphe non-orienté est **connexe** si deux sommets distincts sont toujours reliés par une chaine.

Une **composante** **connexe** d’un graphe non-orienté est un sous-graphe connexe maximal (au sens de l’inclusion).

Un graphe orienté est **fortement connexe** si pour chaque couple de sommets il existe un chemin allant du premier au second (tout sommets vers tout autre sommets).

Une **composante fortement connexe** d’un graphe orienté est un sous-graphe connexe maximal (au sens de l’inclusion).

Prop :

Deux composantes (fortement) connexes distinctes sont disjointes .

Graphe réduit

Def :

Pour tout graphe orienté G, on définit G’ le **graphe des composantes fortement connexes** de g, ou **graphe réduit** de G, de la façon suivante :

Les sommets de G’ correspondent aux composantes fortement connexes de G ;

Il existe dans G’ un arc de la composante 1 à la composante B !=A si il existe un arc (u,v) dans G, avec u nu sommet de 1 et v un sommet de B.

Propriété :

Un graphe réduit est acyclique

Preuve :

Par l’absurde A1->…->An->A1 Dans G’

Il existe u1…un (appartient à) A1\*…\*An ,u1’ tels que

u1->…->un->u1’->\*u1 (car u1 et u1’ sont dans la même composante connexe)

Montrons que A1 U {u2} est fortement connexe (ce qui est absurde car A1 plus grand au sens de l’inclusion)

Soit u dans A1

u->\*u1->u2 (car A1 fortement connexe)

u2->\*u1->\*u (car cycle et A1 fortement connexe)

Algorithme de Kosaraju

Exercice : les arbres de l’arborescence renvoyée par un parcours de graphe orienté correspondent-ils toujours aux composantes fortement connexes ?

Principe de l’algorithme :

L’algorithme de Kosaraju est un raffinement du parcours en profondeur qui renvoie les composantes fortement connexes sous forme d’arborescence. On procède de la façon suivante :

* Un premier parcours en profondeur renvoie la liste construite sur le même principe que dans le tri topologique ;
* On parcourt ensuite en profondeur le graphe **transposé** de G, en prenant les sommets initiaux dans l’ordre de la liste précédente, et on renvoie l’arborescence générée.

Kosaraju (G)

(\*premier parcours\*)

Soit L = tri\_topologique(G)

(\*second parcours\*)

Soit Gt = transposé(G)

Les sommets de Gt sont non marqués

Les sommets sont leur propre père

Pour chaque sommet s de L

Si s est non marqué

Visiter (s)

Renvoyer P

Visiter(u)

On marque u

Pour chaque v successeur de u

Si v est non marqué

Le père de v devient u

Visiter (v)

Retour sur l’algorithme de Dijkstra

Calcul de distance par un parcours en largeur

Un parcours en largeur permet de calculer la distance de tous les sommets accessibles depuis un sommet source s dans un graphe pondéré :

Distance\_largeur(G,s)

Les sommets sont marqués

On marque s

Soit F une file vide

On enfile s dans F

La distance D[u] de chaque sommet u est 0

Tant que F est non vide

On défile un sommet de F

Pour chaque v successeur de u

Si v est non marqué

On marque v

D[v] devient D[u]+1

On enfile v dans F

L’algorithme de Dijkstra

L’algorithme de Dijkstra est une généralisation du calcul de plus court chemin par parcours en largeur au cas des graphes pondérés. Il nécessite cependant que tous les poids soient positifs.

L’algorithme prend en entrée le graphe et un sommet source s, et renvoie les distances de s sommets à s, ainsi que des plus courts chemins (sous forme d’arborescence)

L’algorithme s’appuie sur la structure de files de priorité (en défilant d’abord les priorités minimales).

Dans la suite, on notera w (u, v) la valeur du poids d’un arc (u, v).

Pseudocode :

Dijkstra (G,s)

Les distances D[u] sont infinies

D[s] devient 0

Soit F une file de priorité vide

Pour tout sommet u :

Ajouter\_valeur(F, -D[u],u)

Chaque sommet est son propre père

Tant que f est non vide

On defile un sommet u de F

Pour chaque v successeur de u

Soit d’ = D[u] + w(u,v)

Si d’ < D[v]

D[v] devient d’

Modifier\_prio(F,v,-d’)

Le père de v devient u

On renvoie les pères et les distances