Algorithmique : Résumé

Sylvain Julmy

17 décembre 2015

1. Pré-requis et introduction

1.1 Arbre de recherche binaire

Caractéristiques :

- Toutes les clés du sous-arbre gauche sont plus petites ou égales à la clé du parent
- Toutes les clés du sous-arbre droit sont plus grandes que la clé du parent
- Si les deux fils sont vides alors le nœud est appelé "feuille"

- Taille mémoire en $O(n)$
- Profondeur en $O(\log(n))$
- Insérer un élément en $O(\log(n))$
- Recherche en $O(\log(n))$
- Parcours en $O(n)$
- Tri en $O(n \cdot \log(n))$

Algorithme 1 : Rechercher un élément dans un arbre

```
Données : La racine r de l'arbre
Résultat : L'élément recherché
tant que L'élément courant n'est pas l'élément recherché faire
    si La clef de l'élément est plus petite alors
        L'élément courant vaut l'élément de gauche
    sinon
        L'élément courant vaut l'élément de droite
    fin
fin
Retourner l'élément trouvé ;          // Vaut NULL si l'élément n'existe pas
```

Algorithme 2 : Insérer un élément dans un arbre

```
Données : La racine r de l'arbre
Données : L'élément à insérer
Résultat : L'arbre avec l'élément
Rechercher l'élément à insérer ;                          // Vaudra NULL
Remplacer le pointeur vide par l'élément à insérer;
```

Algorithme 3 : Retirer un élément dans un arbre

Données : La racine r de l'arbre
Données : L'élément à retirer
Résultat : L'arbre sans l'élément
Rechercher l'élément à insérer;
si *C'est une feuille* **alors**
| Supprimer la feuille
fin
si *C'est un arbre avec un seule fils* **alors**
| Remplacer l'élément par le fils
fin
si *C'est un arbre avec deux fils* **alors**
| Remplacer l'élément par celui le plus à gauche dans le sous-arbre droit;
| Si l'élément à un fils, le remplacer par le fils; `// N'a jamais deux fils`
fin

Un arbre peut potentiellement se déséquilibrer et se transformer en liste chainée. Solutions :

— Si on connait les éléments à l'avance, les insérer dans un ordre aléatoire (statistiquement c'est bon)
— Imposer un différence maximum de hauteur entre tous les sous-arbres frères.

1.2 Tas binaire

Un **tas max** est un arbre binaire équilibré, complet sur la gauche et où la clé n'est pas plus petite que la clé de ses fils. Le maximum des éléments se trouve donc à la racine.

— Taille mémoire en $O(n)$
— Insertion d'un élément en $O(\log(n))$
— Recherche quelconque en $O(n)$
— Recherche du max en $O(1)$
— Parcours en $O(n)$

Algorithme 4 : Insérer un élément dans un tas

Données : La racine r du tas
Données : L'élément à insérer
Résultat : Le tas avec l'élément
Rechercher l'élément à insérer; `// Dernier élément du parcours en largeur`
Remplacer le pointeur vide par l'élément;
tant que *Le parent est plus grand que l'élément* **faire**
| Inverser le parent avec l'élément courant
fin

Algorithme 5 : Retirer le max dans un tas

Données : La racine r du tas
Résultat : Le tas sans l'élément
Rechercher le dernier éléments du tas par un parcours en largeur;
Supprimer le max;
tant que *L'élément est plus grand qu'un des fils* **faire**
| L'inverser avec le fils le plus grand
fin

1.3 Complexité algorithmique

Soient $f(x)$ et $g(x)$, on dit que f est d'ordre inférieur ou égale à g si : $f(x) \leq c \cdot g(x) \forall x \geq x_0,\ x_0 > 0,\ c > 0$. On écris alors $f \in O(g)$.

- f **est dominé par** g ($f \in O(g)$) si il existe une constante c tel que pour tout $n \geq n_0$ on a $c \cdot g(n) \geq f(n)$
- L'inverse se dit f **domine** g (dans le cas ci-dessus c'est g qui domine f)
- Si f **est borné dessus et dessous** par g : c_1, c_2 où $\forall n \geq n_0$ on a $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

1.4 Théorème général de récurrence

Temps pour résoudre un problème de taille n récursivement : $T(n) = aT(n/b) + f(n)$ avec $a \geq 1$ et $b > 1$. On doit résoudre a sous-problème de même type que le problème initial qui est découpé en b parties. Il faut $f(n)$ pour rassembler b.

1.5 Graphe

- Une boucle est une arête $e = (v_0, v_0)$ incidente au même sommet
- Un sommet isolé est de degré 0
- Un sommet de degré 1 est dit "pendant"

1.5.1 Non-orienté

On note un graphe non-orienté $G = (V, E)$ avec un ensemble V de sommet ($|V| = n$) et un ensemble E d'arête ($|E| = m$). A chaque arête est associé une paire de sommet appelée les extrémités. Deux sommets sont adjacent si il existe une arête entre eux. Une arête qui relie les sommets u et v est dit incidente à u et v.

- Le degré d'un somme v noté $deg(v)$ est le nombre de d'arête incidents à v
- Une chaîne est une suite alternée de sommet et d'arête, commençant et finissant par un sommet. Une chaîne est dit simple si chaque **arête** y apparait une fois au plus. Une chaîne est dit élémentaire si chaque **sommet** y apparait une fois au plus. La longueur

d'une chaîne est égal au nombre d'arête qu'elle contient. Un cycle est une chaîne dont les deux extrémités sont confondus et qui contient au moins une arête.

Le théorème d'Euler nous dit que $\sum_{v \in V} deg(v) = 2 \cdot |E|$ et qu'il y a un nombre pair de sommets de degré pair.

1.5.2 Orienté

On note un graphe orienté $G = (V, E)$ avec un ensemble V de sommet ($|V| = n$) et un ensemble E d'arête ($|E| = m$). A chaque arête est associé une paire de sommet appelée les extrémités. Deux sommets u et v sont adjacent si il existe une arête (u, v) entre eux. u est l'extrémité initiale et b terminale.

- Le degré extérieur $deg^+(v)$ est la somme des arêtes dont l'extrémité initiale est v
- Le degré intérieur $deg^-(v)$ est la somme des arêtes dont l'extrémité terminale est v
- Le degré deg vaut $deg(v) = deg^+(v) + deg^-(v)$
- Un chemin est une chaîne mais dont les orientations concordent
- Un circuit est un cycle avec la même contrainte que ci-dessus

Le théorème d'Euler nous dit que $\sum_{v \in V} deg^+(v) = \sum_{v \in V} deg^-(v) = |E|$

2. Exercices 1

1.1 Il n'est pas possible de réaliser ce problème, si on compte le nombre d'intersection total pour chaque segment, on arrive à $5 * 3 = 15$, or, lors de chaque intersections, cela rajoute à chaque fois 2 intersections, donc on arrive jamais à 15 puisse que 15 est impair.

Avec 301 segments qui doivent en couper exactement 201 autres, cela nous donne $301 * 201 = 60501$, c'est impair dont impossible.

Pour résoudre ce problème on peut modéliser sous la forme d'un graphe : 5 sommets connectés à exactement 3 autres sommets.

1.2 Ajouter des nœuds supplémentaires avec un arbre de Steiner, l'arbre de Steiner ce construit à partir du diagramme de Voronoï.

1.4 Un problème est donnée par une matrice de flot F et une matrice de distance D. Si la matrice D est plus grande que F (si il y a plus de place que d'éléments à placer), alors on peut modifier la matrice F avec des éléments quelconque dont le coût est l'élément neutre.

1.5 $x_i \; in \; \{0, 1\}$ avec $\sum i \cdot x_i = 1170$ et $\prod i \cdot (1 - x_i) = 36000$ et donc on peut définir la fonction d'utilité $min(|\sum i \cdot x_i - 1170| - |\prod i \cdot (1 - x_i) - 36000|)$

1.6 On peut modéliser ce problème comme un problème de coloration de graphe.

1.9

3. Méthodes constructives

3.1 Construction aléatoire

Tirer aléatoirement une solution dans l'espace des solutions admissibles. L'avantage est que la méthode est très facile à implémenter mais la qualité de la solution est déplorable et un tirage aléatoire uniforme n'est pas évident à réaliser.

σ : permutation aléatoire $1..n$
σ_i : ième ville visité
$D = (d_{ij})$
minimiser $(\sum_{i=1}^{n-1} d_{\sigma_i \sigma_{i+1}}) + d_{\sigma_n \sigma_i}$

ou bien minimiser avec s_i est la ville qui suit la ville i

$$\sum_{i=1}^{n} d_{is_i}$$

Données : Tableau de n element L
Résultat : Une permutation aléatoire de L
Définir l comme la longueur du tableau;
pour *i allant de 1 à n* **faire**
    Tirer aléatoirement $j \in [i; n]$;
    Permuter $L[j]$ avec $L[l]$;
    $l = l - 1$;
fin

3.2 Méthode gloutonne

L'idée est de construire une solution élément par élément en ajoutant, à chaque pas, un élément approprié. Cela est optimal pour certain problème.

On part d'une solution s vide ou trivial. On a une fonction de coût incrémental qui mesure empiriquement l'adéquation d'ajouter l'élément e à s. Le fait d'ajouter un élément peut ajouter des contraintes sur les prochains éléments à ajouter.

; `/* Algorithme glouton en` $O(n^2)$ `*/`
Données : Une solution partielle minimal s // en général $\emptyset$
Données : $R = E$ // Ensemble des éléments pouvant être ajoutés à s
Résultat : Une solution gloutonne
tant que s *n'est pas une solution complète* **faire**
  Calculer $c(e, s) \forall e \in R$;
  Choisir un e' optimisant $c(e, s)$;
  $s = s \vee e'$;
  ; `// propagation des contraintes`
  Supprimer de R tout les éléments qui ne peuvent être ajouter à s;
fin

Il existe d'autre algorithme :

3.2.1 Regret maximum

Lors de chaque étape, choisir la ville e qui maximise la fonction

$$c(s, e) = min_{j,k \in R} d_{je} + d_{ek} - min_{j \in R} d_{ie} + d_{ej}$$

3.2.2 Meilleur insertion

Choisir la ville e qui minimise la fonction

$$c(s, e) = \text{coût d'insertion minimalde la ville e avec la tournée partiel s}$$

possible en maximisant : insertion de la ville la plus éloignée. Les deux méthodes sont en $O(n^2)$

4. Méthodes d'amélioration

Pseudo code d'une méthode d'amélioration locale :

```
; /* Trame d'une méthode d'amélioration locale                    */
Données : Une solution donnée (par exemple, à partir d'une construction gloutonne
Résultat : Une solution équivalente ou meilleure
do
    Essayer de trouver une amélioration;
    Faire l'amélioration trouver;
while Une amélioration est effectué;
```

Exemple de modification :

- Remplacer deux arêtes d'une tournée par deux autres
- **2-Opt** : inverser le sens de parcours d'une sous-chaîne (remplacer deux arêtes par deux autres)
- **3-Opt** : déplacer un chemin ailleurs dans la tournée (remplacer trois arcs par trois autres)
- **Or-Opt** : Déplacer une sous-chaîne de r sommet ailleurs dans la tournée avec $r = 3$ puis 2 puis 1 etc...

3.1 On arrive deux fois à -4 pour le premier chemin améliorant.

5. Méthodes aléatoires

5.1 Choix du prochain élément

Il existe plusieurs technique pour ce choix :

- GRASP : on calcule $c_{min} et c_{max}$ (coût d'insertion) et on choisis l'élément parmis un sous-ensemble R de E où E est l'ensemble des éléments disponibles et R est $\{r \in E | c_r \in [c_{min}; \alpha(c_{max} - c_{min})]\}$
- colonie de fourmie : le choix est inversement proportionnel au coût et les coût sont modifiés en fonction des solutions construites précédemment
- technique du bruitage : bruitage du coût en fonction d'une loi

5.2 Recherches locales aléatoires

5.3 Exercices

4.1 :

6. Méthodes de décomposition

En général, utiliser pour résoudre des problèmes de très grandes tailles (entre 10^3 et 10^7 d'ordre de grandeur).

- Jouet : énumération complète
- Petit : Méthodes exactes
- Moyen : Méta-heuristique (limite de la mémoire en $O(n^2)$)
- Grand : Méthode de décomposition
- Très grand : Bases de données distribuées

6.1 Recherche dans de grand voisinage

Cette technique est applicable typiquement dans un problème de programmation par contrainte avec plusieurs millier de variable. L'idée est de fixée les valeurs de toutes les variables sauf un certain sous-ensemble. On optimise le sous-ensemble et on peut recommencer avec un autre sous-ensemble une fois cela fait.

6.2 Popmusic : méthode de décomposition générique

Idée générale de Popmusic :

```
; /* Algorithme Popmusic                                              */
Données : Une solution donnée (par exemple, à partir d'une construction gloutonne
Résultat : Une solution équivalente ou meilleure
pour Chaque sous-ensemble de la solution faire
  | Optimiser le sous-ensemble;
fin
```

La difficulté est que les sous-ensemble ne sont peut-être pas indépendant les uns des autres.

; `/* Algorithme plus précis pour Popmusic */`
Données : Une solution $S = s_1 \vee s_2 \vee ... \vee s_p$
Données : $O = \emptyset$
Résultat : Une solution équivalente ou meilleure
tant que $S \neq O$ **faire**
  Choisir un élément $S_i \notin O$;
  Créer un sous-problème R composée de $r_i \in S$ les plus proches de s_i;
  Optimiser R;
  si R *est amélioré* **alors**
    $O \leftarrow O \setminus R$
  sinon
    $O \leftarrow O \vee s_i$
  fin
fin

; `/* Grand voisinage pour le VRP */`
Données : Une solution S initial
Résultat : Une solution équivalente ou meilleure
Â partir de S, supprimer n point de la solution;
Réinsérer les clients au mieux;

6.2.1 Popmusic pour la classification non-supervisée

Les parties sont les éléments d'une même classe, on calcule la dissimilarité moyenne entre les éléments de classes différentes et la distance entre les centroïdes, puis on optimise la solution en déplacement progressivement des centres et on recalcule le tout avec l'algorithme K-means.

6.2.2 Compression d'image par quantification de vecteur

On cherche à décomposer une image en blocs de b pixels (par exemple $b = 5 \times 3$). On cherche à trouver la meilleur palette de 2^k couleurs, vues comme des vecteurs de $3 \times b$ octets. Il s'agit d'un problème de classification composer de millions d'éléments en des milliers de groupes. On code chaque bloc par k bits.

6.3 Exercices

5.1 :

5.2 :

7. Fourmis artificielles et constructions aléatoires

Algorithme 6 : Système de fourmis pour le TSP

Données : Une matrice de distance entre les villes $D = (d_{ij} = \frac{1}{\eta_{ij}})$
Données : Une matrice de trace $T = (\tau_{ij})$
Données : Les paramètres α, β, ρ, τ_0, Q et max_iter
Résultat : Une solution
Initialisation : Poser $\tau_{ij} = \tau_0$;
pour max_iter *itérations* **faire**
    $R = (r_{ij}) = 0$; `// Renforcement des arêtes`
    pour $k = \{1, ..., m\}$ **faire**
        $L = 0$;
        Choisir une ville i au hasard;
        tant que *Toutes les villes ne sont pas visité* **faire**
            Choisir une ville j non-visitée avec P proportionnelle à $\tau_{ij}^{\alpha} \cdot \eta_{ij}^{\beta}$;
            $L = L + d_{ij}$;
            $i = j$;
        fin
        pour *tout les trajets* (i, j) *de la tournée* **faire**
            $r_{ij} = r_{ij} + Q/L$
        fin
    fin
    $T = (1 - \rho)T + R$; `// Mise à jour des traces de phéromones`
fin

8. Recherche avec tabous

Il s'agit d'une recherche locale avec la politique du meilleur mouvement sauf qu'elle possède certain ajout :

- Interdire de revenir à une solution déjà visité
- Interdire d'effectuer l'inverse d'un mouvement
- Pénaliser les mouvements fréquemment utilisés
- Forcer l'utilisation de mouvement jamais utilisés
- Lever une interdiction après un certain nombre d'itérations

Besoins :

- Une fonction d'aspiration : un mouvement interdits, qui améliore s^*, est accepté
- Intensification de la recherche : diminuer le nombre de mouvement interdits et repartir depuis la meilleure solution trouvée
- Diversification de la recherche : Fréquence des mouvements utilisés, mémoire des bonnes solutions (Chemin de liaison, construction de vocabulaire)

Algorithme 7 : Recherche avec tabous de base

Données : Solution initiale s
Données : Fonction utilité f à minimiser
Données : Ensemble M des mouvements applicables à toute solution
Données : Paramètres t(durée des interdictions), max_iter
Résultat : Une solution équivalente ou meilleure
Initialisation : $T = \emptyset$, $s^* = s$;
pour max_iter **faire**
    meilleur_voisin = ∞;
    pour $m \in M$, $m \notin T$ **faire**
        si $meilleur\_voisin > f(s \oplus m)$ **alors**
            meilleur_voisin = $f(s \oplus m)$;
            meilleur_mouvement = m;
        fin
        $s = s\oplus$ meilleur_mouvement;
        Remplacer dans T le plus ancien mouvement par $meilleur\_mouvement^{-1}$;
        si $meilleur\_voisin < f(s^*)$ **alors**
            $s^* = s$;
        fin
    fin
fin
Retourner s^*;

8.1 Recherche avec tabous pour le QAP

Une solution est une permutation p de n éléments. Mouvement applicable à n'importe quelle : Transposer les objets i et j ($m = (i, j), 1 \leq i < j \leq n$). C'est à dire : l'objet i actuellement en position p_i est déplacé en position p_j et l'objet j est déplacé en p_i.

Inverse d'un mouvement : replacer simultanément l'objet i en p_i et l'objet j en p_j.

Mémoire :
- Matrice $T = t_{ir}$ de taille $n \times n$
- t_{ir} numéro d'itération où l'on peut à nouveau placer l'objet i en position r
- Le mouvement (i, j) est interdit à la solution p à l'itération $k \iff (t_{ip_j} > k) \wedge (t_{jp_i} > k)$
- Le statut d'un mouvement (i, j) peut être tester en temps constant

8.2 Durée des interdictions

La durée des interdictions influence énormément l'évolution de l'algorithme, si la durée est trop faible, on risque de revenir sur ses pas, faire des cycles ou encore d'être piéger dans un optimum local. Au contraire, si elle est trop grande, il y aura un nombre limité de mouvement réalisable et on risque fortement de ne pas voir "les bon chemins" car tout le temps une partie est interdite.

Solutions :
- Apprendre dynamiquement la durée des interdictions (augmenter la durée des interdictions si on visite 2 fois la même solution, diminuer la durée si on ne visite pas la même solution pendant un plusieurs itérations)
- Tirer aléatoirement la durée des interdictions : des mouvements sont interdits pendant longtemps mais la plupart sont juste inaccessible pendant un court instant

8.3 Mémoire à long terme

Pénalité sur les fréquences : Le but est d'éviter les mouvements de faibles coût en mémorisant la fréquence d'utilisation de chaque mouvement et en pénalisant de $F \cdot freq(m)$ le mouvement m.

Recherche locale guidée : Lorsqu'un optimum local est atteint, l'élément le plus couteux de la solution est pénalisé pour défavoriser sont utilisation future.

Forcer des mouvements : Casser la structure d'une solution en identifiant les mouvements jamais choisis durant les K dernières itérations. Pour QAP, on lit cela directement dans la matrice T.