Synthèse des cours 6 et 7 : programmation dynamique (3 et 4)

19 et 26 octobre 2021

François Laroussinie

NB : Ces synthèses ont pour but de compléter les notes prises en cours. Elles ne les remplacent pas ! En particulier, la plupart des preuves n'y figurent pas. Rappel : il faut programmer les algorithmes vus en cours.

1 Plus courts chemins dans un graphe

Etant donné un graphe orienté valué $G = (S, A, w)$, deux algorithmes classiques pour trouver les plus courts chemins sont basés sur la programmation dynamique : l'algorithme de Bellman-Ford et l'algorithme de Floyd-Warshall.

L'algorithme de Bellman-Ford a pour objectif de calculer les distances $d[u]$ (pour tout sommet u) correspondant à la distance d'un PCC entre un sommet s fixé et le sommet u. Initialement $d[u]$ vaut ∞ (sauf $d[s] = 0$). L'algorithme procède en $|S| - 1$ itérations au cours desquelles, pour chaque arc (u, v), on met à jour $d[v]$ avec $\min(d[v], d[u] + w(u, v))$. On peut alors montrer qu'après la i-ème itération $d[v]$ est inférieur ou égal à la distance d'un PCC entre s et v comprenant au plus i arcs. Cela garantit la correction de l'algorithme puisque tout PCC (si il en existe !) comprend au plus $|S| - 1$ arcs. Ici on calcule donc les distances de PCC de longueur (au sens du nombre d'arcs) de plus en plus grande...

L'algorithme de Floyd-Warshall vise à calculer la distance d'un PCC entre n'importe quelle paire de sommets. Ici le graphe est représenté sous forme matricielle. On suppose que $S = \{x_1, \ldots, x_n\}$. La matrice $n \times n$ M_G décrit le poids des arcs : $\alpha_{i,j} = p \in \mathbf{R}$ si $w(x_i, x_j) = p$ et $\alpha_{i,j} = \infty$ si $(x_i, x_j) \notin A$. L'idée ici est de calculer les distances des PCC en autorisant de plus en plus de sommets intermédiaires. L'algorithme commence par les distances des PCC sans sommet intermédiaire, on dit d'*intérieur* vide (donc des PCC de longueur nulle ou égale à un arc). Puis on autorise éventuellement de passer par x_1, puis par x_1 et/ou x_2 *etc.* On retrouve là aussi le schéma de la programmation dynamique avec l'équation :

$$d_{i,j}^{\{x_1,\ldots,x_{k+1}\}} = \min(d_{i,j}^{\{x_1,\ldots,x_k\}}, d_{i,k+1}^{\{x_1,\ldots,x_k\}} + d_{k+1,j}^{\{x_1,\ldots,x_k\}})$$

où $d_{i,j}^{\{x_1,\ldots,x_{k+1}\}}$ représente la valeur d'un PCC entre x_i et x_j ne pouvant passer que par des états intermédiaires dans l'ensemble $\{x_1, \ldots, x_{k+1}\}$.

2 Un jeu

On considère un jeu de cartes à deux joueurs. Au début du jeu, on dispose une suite de n cartes $c_1, \ldots, c_n$ sur une table (face visible). Chaque carte c_i a une valeur $v_i \in \mathbf{R}$. A chaque tour, le joueur 1 choisit une des deux cartes situées à une extrémité (gauche ou droite) de la rangée. Le joueur 2 fait ensuite de même. Le jeu s'arrête lorsque toutes les cartes ont été prises par les deux joueurs. Le gagnant est celui dont la somme des valeurs de ses cartes est la plus grande.

A chaque tour, la position du jeu est définie par une paire (appelée configuration) (i, j) où $1 \leq i \leq j \leq n$: la carte la plus à gauche est c_i et celle la plus à droite est c_j. La configuration initiale est $(1, n)$. L'objectif est de calculer la meilleure stratégie possible pour le joueur 1.

On peut construire une table $\mathsf{TV}[-,-]$ de dimension $n \times n$ telle que pour toute configuration (i, j) $\mathsf{TV}[i, j]$ contient la valeur optimale que peut **assurer** le joueur 1 lorsqu'il a le trait (NB : « assureR » signifie ici que le joueur 2 joue le mieux possible) :

$$\mathsf{TV}[i,j] \;=\; \begin{cases} 0 & \text{si } i > j \\ v_i & \text{si j=i} \\ \max(v_i, v_j) & \text{si j=i+1} \\ \max(v_i + \min(\mathsf{TV}[i+2,j], \mathsf{TV}[i+1,j-1]), & \\ \qquad v_j + \min(\mathsf{TV}[i,j-2], \mathsf{TV}[i+1,j-1])) & \text{sinon} \end{cases}$$

Il reste à énumérer correctement des paires i, j afin de remplir le tableau $\mathsf{TV}[i, j]$... A faire en exercice !

3 Construction d'un ABR optimal

On rappelle que dans un ABR (arbre binaire de recherche), on stocke un élément dans chaque noeud de l'arbre et que l'on assure que si x' est dans le sous-arbre gauche de x alors $x' \leq x$, et si x' est dans le sous-arbre droit de x alors $x' > x$.

L'idée est de construire un ABR optimal à partir d'un ensemble d'éléments dont on connait la fréquence (de recherche). Par optimal, on entend un ABR A tel que le coût total de la recherche des n éléments selon leur fréquence respective soit minimal, c'est-à-dire que la somme $\sum_{x:\text{élt}}((\mathsf{prof}_A(x)+1)\cdot\mathsf{freq}(x))$ soit minimale (où $\mathsf{prof}_A(x)$ désigne la profondeur du noeud de A correspondant à l'élément x et $\mathsf{freq}(x)$ la fréquence de x). Notons que $(\mathsf{prof}(x)+1)$ correspond au coût (en nombre de comparaisons) de la recherche de l'élément x dans l'ABR. Formellement, on a donc :

Input : n valeurs $x_1, \cdots, x_n$ ordonnées $(x_1 \leq \cdots \leq x_n)$ et n fréquences $f_1, \cdots, f_n$.
Output : Un ABR contenant les valeurs x_i minimisant la somme $\sum_{x:\text{élt}}((\mathsf{prof}(x)+1)\cdot\mathsf{freq}(x))$.

Pour résoudre ce problème, on va calculer le coût minimal $d_{i,j}$ (avec $i \leq j$) d'un ABR contenant les éléments $x_i, x_{i+1}, \ldots, x_j$.

La valeur de $d_{i,j}$ va dépendre du choix de l'élément positionné à la racine de l'ABR. Supposons que ce soit l'élément x_k avec $i \leq k \leq j$, alors le sous-arbre gauche (un ABR optimal !) contiendra les éléments $x_i, \ldots, x_{k-1}$, et le sous-arbre droit contiendra les éléments $x_{k+1}, \ldots, x_j$.

On peut montrer (cf le cours) que :

$$d_{i,j} = \sum_{i \leq e \leq j} \mathsf{freq}(x_e) + \min_{i \leq k \leq j}(d_{i,k-1} + d_{k+1,j})$$

avec comme convention $d_{i,i} = \mathsf{freq}(x_i)$ et $d_{i,j} = 0$ si $i > j$. Ensuite il suffit d'énumérer correctement les indices... Voir l'algorithme 1.

Pour obtenir l'ABR optimal, il suffit de stocker la racine choisie pour chaque paire (i, j)...

```
Procédure ABRopt (X, F)
begin
    D : matrice d'entiers de taille |X| × |X|, initialisée avec ∞
    pour i = 0 ... |X| faire D[i, i] = F[i]
    pour d = 1 ... |X| − 1 faire
        pour p = 0 ... |X| − d − 1 faire
            i := p,    j := d + p
            pour k = i ... j faire
                aux := 0
                si i ≤ k − 1 alors aux+ = D[i][k − 1]
                si k + 1 ≤ j alors aux+ = D[k + 1][j]
                D[i][j] := min(D[i][j], aux)
            D[i][j] := D[i][j] + Σ_{i≤u≤j} F[u]
    return D[0, |X| − 1]
```

Algorithme 1 : algorithme pour calculer le coût d'ABR optimal