INF4100
Laboratoire no. 8

19/03/08

(Programmation dynamique)

1. Coefficients binomiaux

L'algorithme pour le calcul des coefficients binomiaux vu en cours utilise une matrice à deux dimensions, donc un espace mémoire de complexité $\Theta(n \times k)$. Modifiez l'algorithme pour qu'il utilise une quantité d'espace mémoire qui soit *asymptotiquement* inférieure, tout en étant aussi efficace (programmation dynamique).

2. Algorithme de Floyd

Soit le graphe, valué et orienté, présenté à la figure 1. Utilisez l'algorithme de Floyd pour calculer la matrice des plus courtes distances entre chacun des points du graphe. Donnez la valeur de chacune des matrices intermédiaires (c'est-à-dire, la valeur de `D` au début, ou à la fin, de chacune des itérations).

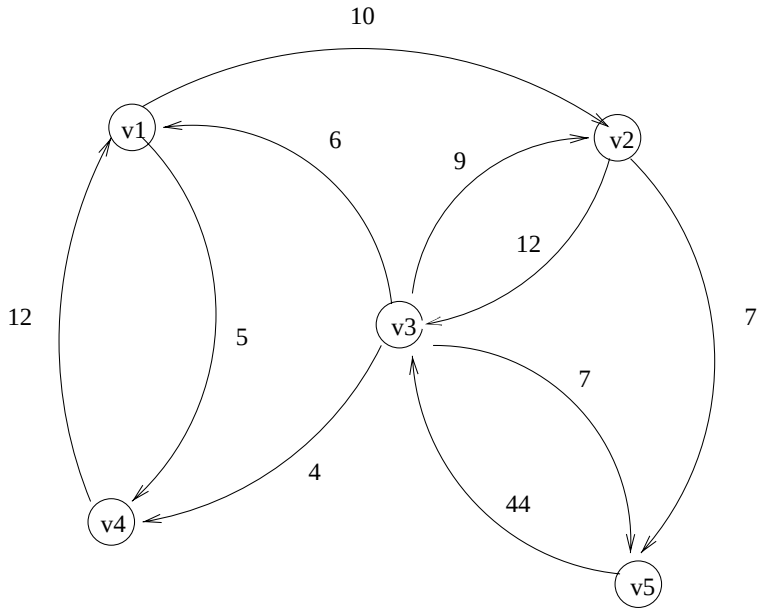

Figure 1: Un graphe valué et orienté avec cinq (5) sommets.

3. Graphe valué non orienté

Un graphe valué non orienté est un graphe où les liens entre les sommets ont un poids, mais où ces liens n'ont pas de direction. En d'autres mots, soit il existe une «arête» entre les deux sommets — auquel cas le poids indique la distance pour aller du premier sommet vers le deuxième ou *vice-versa* — soit il n'en existe pas — auquel cas le poids peut être défini comme étant «$+\infty$» dans une représentation matricielle (matrice d'adjacence). La figure 2 présente un tel graphe.

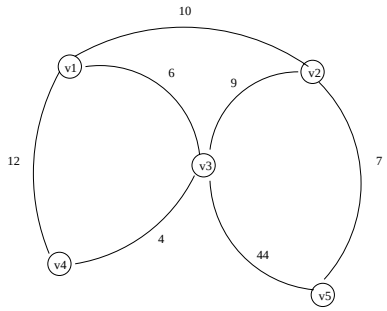

Figure 2: Un graphe valué non orienté avec cinq (5) sommets.

Un graphe valué non orienté peut être représenté par une matrice d'adjacence *symétrique* — dont telle que `G[i,j] = G[j,i]`, pour tout `i` et `j`. Évidemment, la matrice des distances minimums entre les divers sommets est elle aussi une matrice symétrique.

De quelle façon pourrait-on modifier l'algorithme de Floyd pour produire la matrice des distances minimums entre toutes les paires de sommets, mais en ne calculant qu'une partie de la matrice — par exemple, la partie triangulaire supérieure?

4. Distance d'édition

Dans plusieurs langues, les voyelles contiennent de l'information redondante. Ainsi, il est relativement facile de lire et comprendre le texte suivant : «*En d'trs mts, il st gnrlmnt pssibl d lir t cmprndr n txt, et c mêm s l plprt ds vylls nt été spprms.*»

En termes de distance d'édition, on pourrait donc considérer que le coût d'ajouter ou supprimer une voyelle, ou remplacer une voyelle par une autre, est inférieur au coût pour faire les mêmes opérations lorsque des consonnes sont impliquées.

Pour cet exercice, on va donc supposer que le coût pour insérer, supprimer ou substituer une voyelle par une autre voyelle est d'une (1) unité. Par contre, le coût pour insérer, supprimer ou substituer une consonne par un autre caractère — qu'il s'agisse d'une consonne ou voyelle — sera plutôt de deux (2) unités.

a. Quelles modifications faudrait-il apporter aux fonctions de l'exercice sur la distance d'édition pour produire cette variante de distance d'édition?

b. Quelle serait alors la distance entre les mots "`chien`" et "`chats`"?

5. Sac à dos 0–1 (Pas facile!)

Pour trouver le bénéfice optimal dans le problème du sac à dos 0–1, l'algorithme de programmation dynamique produit une matrice `M` de taille $(n+1) \times (W+1)$. La solution optimale est alors celle définie à la position `M`$[n, W]$.Or, dans de nombreux cas, on pourrait ne remplir que certaines des cases de la matrice, c'est-à-dire celles correspondant effectivement à des sous-problèmes pour lesquels on doit mémoriser la solution. Disons d'un tel remplissage qu'il s'agit d'un *remplissage paresseux*.

Écrivez (en pseudo-code) une version itérative de l'algorithme de programmation dynamique pour le sac à dos 0–1, mais qui remplit *de façon paresseuse* la matrice de calcul des coûts. Vous pouvez supposer que la matrice `M` est déclarée comme suit, donc initialisée avec des 0 partout :

```
int M[0:n,0:W] = ([n+1] ([W+1] 0));
```

Suggestion : Une façon possible est de simuler les appels faits par une version récursive, et ce en utilisant une pile. Vous pouvez supposer que le pile peut recevoir des *paires*, par exemple :

```
p <- new Pile()
p.empiler( (n, W) );
...
(k, w) <- p.sommet();
...
p.depiler();
...
```