ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Section d'Informatique et de Systèmes de Communication

Corrigé de la série 9 23 Novembre 2007

1. Recherche dans un labyrinthe

a) Il y a plusieurs manières de faire. On peut modifier la variante du cours pour mémoriser sur le stack aussi le chemin de retour. Une manière plus simple de procéder est d'utiliser un stack implicitement via récursion.

Algorithme 1 DfsRecursive

Input: s sommet de départ dans un graphe non marqué
Output: Traverse le graphe en DFS.

```
Marquer s
Print "Arrivé à " s
if s est la sortie then
    Print "Sortie."
    return true
for v ∈ neigh(s) do
    if v non marqué then
        found ← DfsRecursive(v)
        if found = true then
            return true
        Print "De retour sur " s
return false
```

b) Soit e l'arête $x \to y$. À l'aller, sur un sommet on a toujours la condition que x est marqué et y pas. Immédiatement en arrivant sur y on le marque, ce qui fait que e ne peut être plus jamais réutilisé à l'aller.

Si e est utilisé à l'aller, e est réutilisé au retour (à moins que la sortie du labyrinthe ne soit trouvée avant). À chaque retour correspond exactement un aller, ce qui fait que l'arête e est utilisée deux fois au plus.

c) L'ensemble des arêtes du chemin ne contient pas de cycle (comme DFS ne visite aucun sommet deux fois), et donc le nombre d'arêtes qui apparaît dans le chemin est $\leq n-1$. Ces arêtes seront utilisées au plus deux fois, à l'exception de la dernière arête devant la sortie qui est utilisée au plus une fois (car une fois que la sortie est atteinte, on s'arrête), ce qui donne un chemin de longueur au plus $2(n-2)+1 = 2n-3$.

d) Considérons le graphe à n sommets suivant. $V = \{s, v_1, \ldots, v_{n-1}\}$. $E = \{(x,y) \mid x = s \vee y = s\}$. C'est une étoile. Montrons que si l'on part de s, un chemin de longueur 2ℓ visite $\ell+1$ sommets au plus. On procède par récurrence. Si $\ell = 0$, c'est clair : on ne visite que s. Sinon, après un déplacement, on se trouve sur un des v_i, et le deuxième déplacement retourne forcément vers s. Le reste du chemin est de longueur $2\ell - 2 = 2(\ell-1)$, et l'hypothèse d'induction dit alors que ce bout de chemin visite au plus ℓ sommets. Donc, le chemin total visite au plus $\ell+1$ sommets.

Comme il y a n sommets dans le graphe, on en visite alors au plus $n-1$ avec un chemin de longueur $2n-4 = 2(n-2)$.

e) Un chemin de longueur $\leq n-2$ ne peut visiter au plus $n-1$ sommets, mais il y en a n.

2. Traverser des graphes

a) 1,12,4,6,9,10,11,3,2,5,8,7 (cette solution n'est pas unique, puisque l'algorithme n'est pas deterministe).

b) 1,12,5,8,4,7,6,2,9,10,11,3 (cette solution n'est pas unique).

- On commence par mettre 1 dans la queue Q.
- On ajoute tous ses voisins non-marqués à Q : 12, 5, 8 (qu'on marque aussi dans cet ordre). On a donc $Q = (1, 12, 5, 8)$.
- On fait dequeue, on a donc : $Q = (12, 5, 8)$.
- On prend le premier élément de Q : 12. On ajoute tous ses voisins non marqués : 4 (on marque aussi 4). On a donc $Q = (12, 5, 8, 4)$.
- On fait dequeue, on a donc : $Q = (5, 8, 4)$.
- On prend le premier élément de Q : 5. On ajoute tous ses voisins non marqués à Q : il n'y en a pas.
- On fait dequeue, donc $Q = (8, 4)$.
- On prend le premier élément de Q : 8. On ajoute tous ses voisins non marqués à Q : 7. On a donc $Q = (8, 4, 7)$.
- Dequeue, donc $Q = (4, 7)$.
- On prend le premier élément de Q : 4. On ajoute tous ses voisins non marqués à Q : $6, 2$. On a donc $Q = (4, 7, 6, 2)$.
- Dequeue, donc $Q = (7, 6, 2)$.
- On ajoute tous les voisins non marqués de 7 à Q : $9, 10$. On a donc $Q = (7, 6, 2, 9, 10)$.
- Dequeue, donc $Q = (6, 2, 9, 10)$.
- On ajoute tous les voisins non marqués de 6 à Q : aucun.
- Dequeue, donc $Q = (2, 9, 10)$.
- On ajoute tous les voisins non marqués de 2 à Q : aucun.
- Dequeue, donc $Q = (9, 10)$.
- On ajoute tous les voisins non marqués de 9 à Q : aucun.
- ...etc...

c) Nous modifions BFS comme suit (les nouvelles lignes sont les lignes 3 et 12) :

Call : BFSDist(G, s)
Input: Graphe $G = (V, E)$, sommet $s \in V$.
Output: Associer à chaque sommet v une valeur $v.dist$ qui représente la distance de s à v.

```
1: Enqueue(Q, s)
2: Marquer s
3: s.dist ← 0
4: while Not QueueEmpty(Q) do
5:     v ← Head(Q)
6:     while N[v] ≠ ∅ do
7:         Choisir v' ∈ N[v]
8:         N[v] ← N[v] \ {v'}
```

```
 9:       if v' n'est pas marqué then
10:         Enqueue(Q, v')
11:         Marquer v'
12:         v'.dist ← v.dist + 1
13:       end if
14:     end while
15:     Dequeue(Q)
16: end while
```

d) Nous voulons le nombre de sommets atteignables depuis s. Nous supposons que s lui-même est atteignable depuis s, nous voulons donc la taille de la composante connexe de s. Nous modifions AbstractTraversal comme suit (les nouvelles lignes sont 3,12 et 17) :

Call : AbstractTraversalCount(G, s)
Input: Graphe $G = (V, E)$, sommet $s \in V$.
Output: Le nombre de sommets atteignables depuis s.

```
 1: Q ← {s}.
 2: Marquer s.
 3: count ← 1
 4: while Q ≠ ∅ do
 5:   Choisir v ∈ Q.
 6:   while N[v] ≠ ∅ do
 7:     Choisir v' ∈ N[v]
 8:     N[v] ← N[v] \ {v'}
 9:     if v' n'est pas marqué then
10:       Q ← Q ∪ {v'}
11:       Marquer v'
12:       count ← count + 1
13:     end if
14:   end while
15:   Q ← Q \ {v}
16: end while
17: return count
```

En effet, il suffit de compter le nombre de sommets qui sont visités lors d'un parcours du graphe commençant à 1.

3. L'algorithme de Dijkstra

a) Appliquons l'algorithme de Dijkstra à ce graphe, pour des raisons de clarté et de comprehension de l'evolution de l'algorithme, il est indiqué le prédécesseur de chaque sommet pour une plus courte distance spécifique trouvée dans le tableau qui suit. Nous soulignons qu'il s'agit d'une indication non obligatoire.

La première colonne indique le nombre d'itérations (voir boucle while comprise entre la ligne 5 à 16 de l'algorithme dans le cours). Pour ce graphe, l'algortihme se termine au bout de 10 itérations.

La deuxième colonne indique le sommet v à chaque fois que on est au début du while (ligne 5 de l'algorithme dans le cours). En gras, nous listons les sommets, chaque ligne correspond à une itération ; le contenu de chaque ligne indique les distances minimales depuis le sommet 1 vers chaque sommet de V. La dernière ligne (Fin de l'algorithme) du tableau ci-dessous indique la distance la plus courte entre 1 et chacun des sommets de V.

		Distance la plus courte (prédécesseur)											
Itér.	v	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
0	1	0	∞	∞	∞	6(1)	∞	∞	6(1)	∞	∞	∞	1(1)
1	12	0	∞	∞	3(12)	6(1)	∞	∞	6(1)	∞	∞	∞	1(1)
2	4	0	4(4)	∞	3(12)	6(1)	13(4)	∞	6(1)	∞	∞	∞	1(1)
3	2	0	4(4)	∞	3(12)	6(1)	6(2)	∞	6(1)	5(2)	∞	∞	1(1)
4	9	0	4(4)	∞	3(12)	6(1)	6(2)	∞	6(1)	5(2)	6(9)	∞	1(1)
5	5	0	4(4)	∞	3(12)	5(2)	6(2)	∞	6(1)	5(2)	6(9)	∞	1(1)
6	6	0	4(4)	∞	3(12)	5(2)	6(2)	∞	6(1)	5(2)	6(9)	∞	1(1)
7	8	0	4(4)	∞	3(12)	5(2)	6(2)	∞	6(1)	5(2)	6(9)	∞	1(1)
8	10	0	4(4)	∞	3(12)	5(2)	6(2)	∞	6(1)	5(2)	6(9)	9(10)	1(1)
9	11	0	4(4)	11(11)	3(12)	5(2)	6(2)	∞	6(1)	5(2)	6(9)	9(10)	1(1)

b) Nous voulons que l'algorithme nous fournisse pour chaque sommet v le plus court chemin du sommet de départ s à v et la liste des sommets qui font partie du chemin obtenu. Nous modifions DIJKSTRA(G, s) (les nouvelles lignes sont 3 et 13) comme suit :

Call : DIJKSTRA(G, s)
Input: Graphe orienté $G = (V, E)$ avec fonction de poids c: $E \to \mathbb{R}^+$, sommet $s \in V$.
Output: pour tout $v \in V$ le plus court chemin du sommet de départ s à v.

```
1:  for v ∈ V \ {s} do
2:      ℓ(v) ← ∞
3:      pred(v) = ∅
4:  end for
5:  ℓ(s) ← 0, T ← ∅, v ← s
6:  while ℓ(v) ≠ ∞ do
7:      T ← T ∪ {v}
8:      for w ∈ V \ T do
9:          if (v, w) ∈ E then
10:             d ← ℓ(v) + c(v, w)
11:             if d < ℓ(w) then
12:                 ℓ(w) = d
13:                 pred(w) = v
14:             end if
15:         end if
16:     end for
17:     v ← arg min_{w∈V\T} ℓ(w)
18: end while
```

Avec la version modifiée de l'algorithme de Dijkstra, les indications concernant les prédécesseurs (entre parenthèses dans le tableau) sont obligatoires pour pouvoir fournir pour

chaque sommet v le plus court chemin de départ s à v.

Pour ce graphe, s vaut 1 et v vaut $2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$. En parcourant la dernière ligne de notre tableau en suivant les colonnes correspondant aux prédécesseurs, on remarque que :

(a) Le plus court chemin de 1 à 2 (qui correspond à une distance de 4) est $1 \to 12 \to 4 \to 2$.

(b) Le plus court chemin de 1 à 3 (qui correspond à une distance de 11) est $1 \to 12 \to 4 \to 2 \to 9 \to 10 \to 11 \to 3$.

(c) Le plus court chemin de 1 à 4 (qui correspond à une distance de 3) est $1 \to 12 \to 4$.

(d) Le plus court chemin de 1 à 5 (qui correspond à une distance de 6) est $1 \to 5$.

(e) Le plus court chemin de 1 à 6 (qui correspond à une distnace de 6) est $1 \to 12 \to 4 \to 2 \to 6$.

(f) Depuis 1, il n'est pas possible d'atteindre le sommet 7, d'où la valeur de ∞ dans la dernière ligne correspondant au sommet 7 du tableau précédent.

(g) Le plus court chemin de 1 à 8 (qui correspond à une distance de 6) est $1 \to 8$.

(h) Le plus court chemin de 1 à 9 (qui correspond à une distance de 5) est $1 \to 12 \to 4 \to 2 \to 9$.

(i) Le plus court chemin de 1 à 10 (qui correspond à une distance de 6) est $1 \to 12 \to 4 \to 2 \to 9 \to 10$.

(j) Le plus court chemin de 1 à 11 (qui correspond à une distance de 9) est $1 \to 12 \to 4 \to 2 \to 9 \to 10 \to 11$.

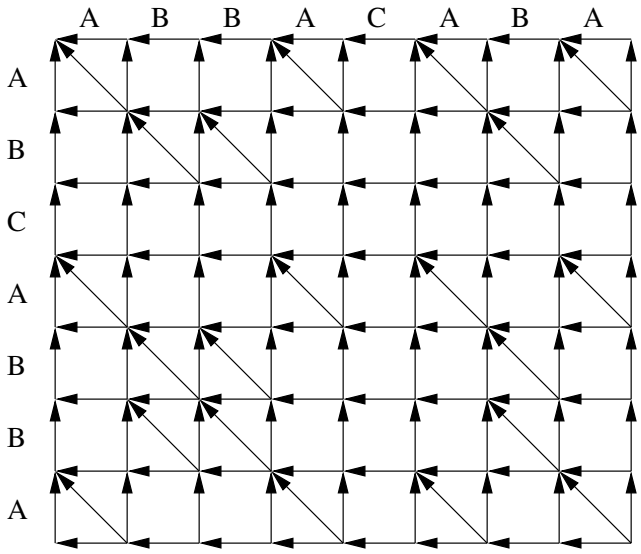
(k) Le plus court chemin de 1 à 12 (qui correspond à une distance de 1) est $1 \to 12$.

4. Dijkstra et Programmation Dynamique

a) On utilise le graphe du corrigé de la série 7, exercice 1. On attribue des poids *huge* aux arêtes verticales et $huge - v_i$ aux arêtes diagonales de la i-ème ligne. (*huge* est choisi de sorte que tous les poids soient positifs.)

Il est alors facile de voir qu'un plus court chemin correspond bien à une solution optimale et que son poids sera $n \cdot huge - v_{\text{opt}}$.

b) On fait un cadrillage d'arêtes de poids un, et on ajoute des diagonales de poids zéro là où on a une correspondance dans les suites. Dessin-exemple :



Le plus court chemin d'en bas à droite jusqu'en haut à gauche est celui qui passe par le nombre maximal de diagonales, ce qui nous donne aussi une sous-suite optimale.

c) Pour le point a) : Si on a un poids maximal pour le knapsack, par niveau on a à peu près $2m$ arêtes. On a n niveaux, donc à peu pres $2mn = O(mn)$ arêtes, ce qui correspond à la complexité de l'algo DP.

Pour le b) : Par nœud du cadrillage, on a au plus 3 arêtes entrantes, donc si on a des sous-suites de longueur m et n, on trouve un nombre d'arêtes $O(mn)$, ce qui correspond à la complexité du premier algo LCS.

Remarquons que ces correspondances ne sont pas des hasards : chaque arête dans le graphe correspond à une décision que l'on prend dans l'algo DP correspondant.

d) Dans ce cas il n'est pas aussi simple de réduire le problème à une application de l'algorithme de Dijkstra. La différence structurelle est que dans ce cas, pour trouver une solution optimale à un sous-problème, on utilise les solutions de *plusieurs* sous-problèmes optimaux.

Plus concrètement, considérons le parenthèsement optimal de $M_i \cdots M_j$. La solution optimale est de la forme $(M_i \cdots M_k)\cdot(M_{k+1} \cdots M_j)$ pour un k, ou $M_i \cdots M_k$ et aussi $M_{k+1} \cdots M_j$ sont des solutions optimales (donc on en réutilise deux !).

Par opposition, si on considère par exemple le sac à dos pour n objets et poids m, on utilise le fait que la meilleure solution est de la forme

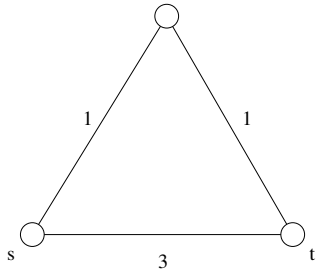
$$S_n(m) = \begin{cases} \text{soit} & \{n\} \cup S_{n-1}(m - c_n), \\ \text{soit} & S_{n-1}(m), \end{cases}$$

donc la solution optimale S_n n'utilise qu'une seule solution optimale de S_{n-1}.

5. Dijkstra et Moore-Bellman-Ford

a) Sous la condition que la constante soit positive, oui. En effet, la longueur de chaque chemin sera multipliée par la même constante, ce qui fait que le chemin le plus court reste le même.

b) Non, pas en général. Considérons un exemple simple :



Le plus court chemin de s à t passe évidemment par le troisième sommet dans ce cas. Si on ajoute 100 à chacun des poids, ça ne reste pas vrai.

c) Le temps de parcours de l'algorithme de Dijkstra est en quelque sorte borné par la longueur du chemin trouvé, ce qui n'est pas le cas pour MBF. Prenons un immense graphe avec seulement des poids 1 sur les arêtes et s et t à (petite) distance fixe. MBF fait n itérations en passant par toutes les arêtes, tandis que Dijkstra ne regarde que les sommets qui sont à distance plus petite que celle entre s et t dans le graphe.

Un exemple concret, c'est le graphe complet.