ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Section d'Informatique et de Systèmes de Communication

Corrigé de la série 9 29 Novembre 2010

1. 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: $s.dist \leftarrow 0$

```
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.

2. L'algorithme de Dijkstra

a) Appliquons l'algorithme de Dijkstra à ce graphe, pour des raisons de clarté et de com-

prehension 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** $v \in V \setminus \{s\}$ **do**
2: $\quad \ell(v) \leftarrow \infty$
3: $\quad pred(v) = \emptyset$
4: **end for**
5: $\ell(s) \leftarrow 0$, $T \leftarrow \emptyset$, $v \leftarrow s$
6: **while** $\ell(v) \neq \infty$ **do**
7: $\quad T \leftarrow T \cup \{v\}$
8: $\quad$ **for** $w \in V \setminus T$ **do**
9: $\quad\quad$ **if** $(v, w) \in E$ **then**
10: $\quad\quad\quad d \leftarrow \ell(v) + c(v, w)$
11: $\quad\quad\quad$ **if** $d < \ell(w)$ **then**
12: $\quad\quad\quad\quad \ell(w) = d$

13: $pred(w) = v$
14: **end if**
15: **end if**
16: **end for**
17: $v \leftarrow \arg\min_{w \in V \setminus T} \ell(w)$
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 $1 \to 12 \to 4 \to 2 \to 9 \to 10$.

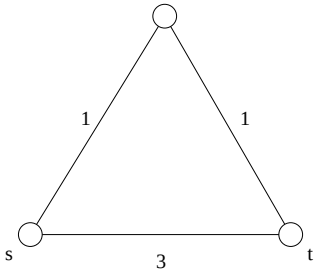
(j) Le plus court chemin de 1 à 11 (qui correspond à une distance de 9) est $1 \to 12 \to 4 \to 2 \to 9 \to 10 \to 11$.

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

3. 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.

4. Priority Queues

a) Ce sont les 15 arrangement suivants:

$$\begin{array}{lll} (4,2,5,1,3), & (4,1,5,3,2), & (4,3,5,1,2), \\ (5,3,4,1,2), & (5,2,4,1,3), & (5,1,4,3,2), \\ (3,5,4,1,2), & (1,5,4,3,2), & (2,5,4,1,3), \\ (1,3,4,5,2), & (2,1,4,5,3), & (3,1,4,5,2), \\ (1,2,4,3,5), & (3,2,4,1,5), & (2,3,4,1,5) \end{array}$$

b) L'idée est de mémoriser dans la queue de priorité les non-premiers et de ainsi simuler un crible d'Erastosthène. On peut alors successivement tester si un nombre donné est premier en testant s'il est le prochain élément dans la queue de priorité. Une première version de cet algorithme serait alors la suivante. (Dans l'algorithme suivant, H est une queue de priorité stockant des entiers, et tel que l'élément prioritaire $\top[H]$ est toujours l'élément le plus petit.

Input: Un entier N indiquant la fin de la queue de priorité
Output: Une suite croissante de tous les premiers entre 1 et N

```
H ← {4, 6, 8, ..., ⌊N/2⌋ · 2}
print 2
i ← 3
while i ≤ N do
    x ← ⊤[H]
    if x > i then
        for j = 2 · i, 3 · i, ..., ⌊N/i⌋ · i do
            H ← H ∪ {j}
        print i
```

else
while $x = i$ **do**
pop(H)
$x \leftarrow \top[H]$
$i \leftarrow i + 1$

Le désavantage de cette solution est qu'il est très couteux de stocker tous les multiples d'un premier dans la queue de priorité. Une petite refléxion montre qu'il suffit de stocker le prochain multiple d'un premier donné si l'on sait de quel premier c'est un multiple: à chaque fois qu'un tel nombre est enlevé de la queue de priorité, il suffit de rajouter le prochain multiple du même premier dans la queue. Pour pouvoir le faire, il faut stocker des couples (x, p), où p est le premier en question est x est son prochain multiple. L'algorithme suivant réalise cet approche avec deux modifications en plus: Les nombres pairs ne sont pas considérés, et les premiers $> \sqrt{N}$ ne sont pas inserés dans la queue puisque les nombres composés $\leq N$ ont toujours un facteur $\leq \sqrt{N}$.

Input: Un entier N indiquant la fin de la queue de priorité
Output: Une suite croissante de tous les premiers entre 1 et N
$H \leftarrow \{(9, 3)\}$
print 2
print 3
for $i = 3, 5, 7, 9, \ldots, \lfloor N/2 \rfloor \cdot 2$ **do**
$(x, p) \leftarrow \top[H]$
if $x > i$ **then**
print i
if $i \leq \sqrt{N}$ **then**
$H \leftarrow H \cup \{(3 \cdot i, i)\}$
else
while $x = i$ **do**
pop(H)
$H \leftarrow H \cup \{(i + 2p, p)\}$
$(x, p) \leftarrow \top[H]$

c) L'opération SiftUp réalise le cœur d'une insertion: il suffit d'ajouter l'élément à la fin et de faire un SiftUp par la suite:

Input: Un heap (a, N), où a est un tableau, et N le nombre d'éléments dans le heap. Un élément e à insérer.
Output: (a, N) modifiés.
$a[N] \leftarrow e$
$N \leftarrow N + 1$
SiftUp$(a, N - 1)$

De manière similaire, on peut aussi effacer un élément en utilisant les opérations SiftUp et SiftDown. Néanmoins, la procédure correcte est légèrement plus compliquée que pour une insertion.

Input: Un heap (a, N) et un indice k d'un élément à enlever.
Output: (a, N) modifiés.
$a[k] \leftarrow a[N - 1]$
$N \leftarrow N - 1$

```
if k < N then
  if a[⌊k/2⌋] < a[k] then
    SiftUp(a, k)
  else
    SiftDown(a, k)
```