Informatique *TP*
Graphes Ipesup

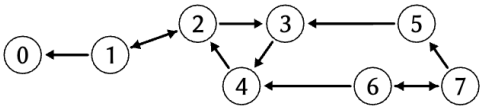
0.1 Exercice

Écrire une fonction qui construit le *graphe miroir* d'un graphe orienté, c'est-à-dire un graphe ayant les mêmes sommets et des arcs inversés.

1 Parcours en profondeur et en largeur

1.1 Exercice

Dérouler à la main le parcours en profondeur sur le graphe suivant, en partant du sommet 3.



1.2 Exercice

Réécrire le parcours en profondeur sans utiliser de récursivité. Pour cela, utiliser une pile contenant des sommets.

1.3 Exercice

Écrire une fonction `has_cycle : digraph -> int -> bool` qui détermine s'il existe un cycle dans un graphe orienté, accessible à partir du sommet donné. Pour cela, utiliser un parcours en profondeur en marquant les sommets avec trois couleurs : non visité / en cours de visite / visité. Si on arrive sur un sommet en cours de visite, c'est qu'on a découvert un cycle. Discuter ensuite le cas d'un graphe non orienté.

1.4 Exercice

Tel que le parcours en profondeur est écrit en classe, il détermine s'il existe un chemin entre le sommet source et tout autre sommet, mais il ne renvoie pas de tel chemin lorsqu'il existe. Pour y remédier, écrire une variante de ce programme qui renvoie un tableau donnant, pour chaque sommet v, le sommet qui a permis de l'atteindre pendant le parcours, le cas échéant, et la valeur -1 sinon. Pour le sommet source, on indiquera sa propre valeur. Écrire ensuite une fonction qui reconstruit le chemin, comme une liste de sommets, entre la source et un sommet donné.

1.5 Exercice

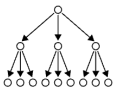
En utilisant un parcours en profondeur, écrire une fonction

```
is_bipartite : graph -> bool
```

qui détermine si un graphe est biparti. Indication : marquer les sommets visités avec une couleur 0 ou 1.

1.6 Exercice

On considère un graphe qui a la forme d'un arbre où le branchement $b \geq 2$ est fixe et où toutes les feuilles sont à la même profondeur h. Voici un exemple avec $b = 3$ et $h = 2$.



On s'intéresse à des parcours de ce graphe depuis la racine, et notamment à leur complexité en fonction de h, le branchement b étant considéré comme une constante.

1. Donner le nombre V de sommets et le nombre E d'arcs de ce graphe, en fonction de b et h.
2. Donner la complexité en espace d'un parcours en profondeur et d'un parcours en largeur de ce graphe.
3. Pour parcourir ce graphe en largeur, on propose cette alternative au programme 8.6 : réaliser $h + 1$ parcours en profondeur successifs, à des profondeurs de plus en plus grandes. Ainsi, le premier parcours en profondeur visite le sommet à profondeur 0, le deuxième parcours visite les sommets à profondeur 1, etc., jusqu'à un dernier parcours en profondeur qui visite les sommets à profondeur h. On appelle cela le *parcours en profondeur itéré* (IDS pour *Iterative Deepening Search*). Donner les complexités en temps et en espace de cette solution. Les comparer aux résultats précédents.

Plus court chemin

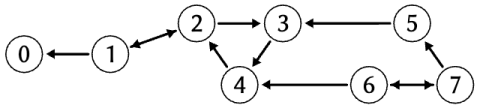
1.7 Exercice

Modifier le Floyd–Warshall pour qu'il renvoie, en plus de la matrice donnant les distances, une seconde matrice donnant, pour chaque paire de sommets (i, j), le sommet k qui a permis l'affectation à la ligne 10. S'il y a un arc entre i et j, mais pas de tel k, alors on prendra (arbitrairement) la valeur i. Et s'il n'y a pas de chemin entre i et j, on prendra la valeur -1.

Écrire également une fonction `print_path` qui affiche le chemin entre deux sommets à partir de l'information contenue dans cette matrice.

1.8 Exercice

Dans cet exercice, on cherche à calculer le nombre de chemins dans un graphe, entre deux sommets i et j donnés, qui ont exactement une longueur k. Ainsi, dans le graphe suivant, il y a 11 chemins de longueur 10 entre les sommets 7 et 2.



Pour faire un tel calcul, on peut se donner une matrice M d'entiers 0 et 1 qui représente l'adjacence du graphe. Ainsi,

$$M_{i,j} = 1 \quad \text{si et seulement s'il y a un arc } i \to j \text{ dans le graphe.}$$

Montrer alors que la matrice M^k détermine exactement le nombre de chemins de longueur k entre deux sommets donnés. En supposant que l'on calcule M^k avec une exponentiation rapide, donner la complexité de cette approche, en fonction de la taille V du graphe et de k. Calculer le nombre de chemins de longueur 42 entre les sommets 7 et 2 dans le graphe ci-dessus.

1.9 Exercice

Dérouler l'algorithme de Dijkstra, à partir de la source 2, en détaillant les opérations faites sur la file de priorité et le contenu du tableau `dist`.

1.10 Exercice

Modifier le programme pour qu'il renvoie, en plus des distances, un tableau donnant, pour sommet w, le sommet v qui a permis de l'atteindre par un plus court chemin. S'il n'y a pas de chemin jusqu'à w, on prendra la valeur -1.

Écrire également une fonction `print_path` qui affiche le chemin entre la source et un sommet donné à partir de l'information contenue dans ce tableau.

1.11 Exercice

Montrer que, si l'heuristique n'est pas admissible, l'algorithme A* peut renvoyer un résultat incorrect, c'est-à-dire un chemin qui n'est pas le plus court.

1.12 Exercice

Quel est le comportement de l'algorithme A* lorsque l'heuristique est définie par $h(v) = 0$ pour tout sommet v ?

1.13 Exercice

Quel est le comportement de l'algorithme A* lorsque l'heuristique est parfaite, c'est-à-dire lorsque $h(v)$ est exactement la distance qui sépare v de la destination dans le graphe ? On pourra supposer qu'il y a un unique plus court chemin de la source à la destination.