Aix*Marseille université

Domaine Sciences et Technologies
Portail 1 René Descartes

Science informatique : TD 3

Année 2023–24

Tris et parcours de graphes

Exercice 1 On a vu le tri par fusion utilisant la méthode *diviser pour régner* :

```
def tri_fusion(t) :
    n = len(t)
    if n <= 1:
        return t
    else:
        return fusionner(tri_fusion(t[0 : n//2]),
                         tri_fusion(t[n//2 : n]))
```

1. Proposer le code d'une fonction `fusionner` afin de compléter le code. Elle doit construire la liste fusionnée en insérant un par un les éléments d'une des deux listes données en argument.
2. Montrer que votre fonction termine.
3. Montrer que votre fonction est correcte en donnant un invariant de boucle (on ne demande pas de prouver formellement que celui-ci est effectivement un invariant).
4. Calculer la complexité dans le pire des cas de votre fonction.

Exercice 2 L'algorithme du tri rapide (appelé *quicksort* en anglais) est un autre tri se basant sur des appels récursifs. Il utilise le partitionnement pour trier les données : on choisit un élément dans le tableau (dans cet exercice, on prendra le premier élément du tableau), qu'on nomme *pivot* et on partitionne le tableau en deux sous-tableaux qui contiennent les éléments du tableau autre que l'élément pivot choisi. Dans l'un des sous-tableaux, on place les éléments inférieurs ou égaux au pivot, dans l'autre, les éléments strictement supérieurs au pivot. On utilise alors des appels récursifs pour trier les deux sous-tableaux, puis on renvoie le tableau recomposé.

Implémenter la fonction `quick_sort` permettant de réaliser ce tri.

Exercice 3

1. Comment sont reliées les valeurs $M_{u,v}$ et $M_{v,u}$ de la matrice d'adjacence M d'un graphe (non orienté), pour tous sommets u et v ?
2. Quelle propriété d'un graphe orienté est représentée par le fait que dans sa matrice d'adjacence M, on a pour tout sommet u, $M_{u,u} = 1$?
3. On peut vérifier qu'une matrice d'adjacence est symétrique à l'aide de la fonction suivante :

```
def est_symétrique(M):
    n = len(M)
    for u in range(n):
        for v in range(n):
            if M[u][v] != M[v][u]:
                return False
    return True
```

   L'algorithme recherche une preuve de non symétrie de la matrice (auquel cas l'algorithme s'arrête en plein milieu en retournant `False` dès que possible) : s'il n'a pas trouvé de preuve de non symétrie, c'est que la matrice est symétrique et on renvoie donc `True`.

   Noter qu'on fait deux fois trop de travail dans ce code, puisqu'on teste chaque couple de sommets (u, v) deux fois... Comment modifier le code pour faire mieux, en ne testant qu'une seule fois chaque couple ?
4. Écrire un algorithme qui prend en entrée la matrice d'adjacence d'un graphe orienté, et renvoie le nombre d'arcs dans le graphe.

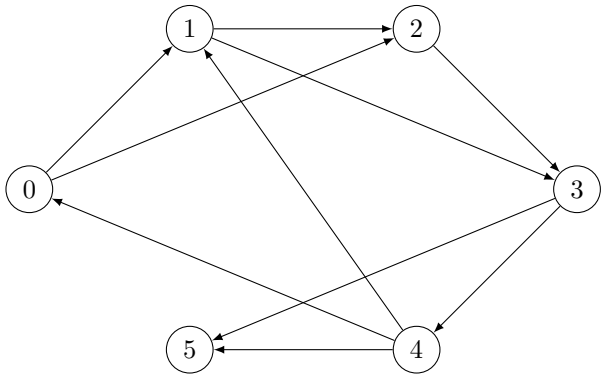
5. Comment modifier votre algorithme pour qu'il compte le nombre d'arêtes d'un graphe non orienté ?

Exercice 4 (Parcours en largeur) L'algorithme de parcours en largeur d'un graphe orienté G permet de visiter tous les sommets accessibles à partir d'un certain sommet s. On peut l'implémenter à l'aide d'une file d'attente des sommets à visiter, ainsi qu'une coloration des sommets visités :

```
def parcours_en_largeur(M, s):
    n = len(M)
    P = nouveau_graphe(n)          #  graphe sans arcs avec n sommets
    F = nouvelle_file()
    couleur = ["blanc"] * n
    couleur[s] = "rouge"
    enfiler(F, s)
    while not est_vide(F):
        u = défiler(F)
        for v in range(n):
            if (M[u][v] == 1) and (couleur[v] == "blanc"):
                couleur[v] = "rouge"
                P[u][v] = 1
                enfiler(F, v)
    return P                       #  graphe des chemins minimaux
```

Cet algorithme retourne la matrice d'adjacence du graphe qui ne contient que les arcs traversés en parcourant un chemin de s à chacun des autres sommets accessibles.

1. Exécuter l'algorithme de parcours en largeur sur le graphe G représenté ci-dessous à partir du sommet $s = 0$, en montrant toutes les étapes de l'algorithme (avec la coloration des sommets et l'état de la file dans les configurations intermédiaires). Représenter aussi le graphe retourné par la fonction.



2. Écrire la fonction `nouveau_graphe` permettant de générer la matrice d'adjacence d'un graphe sans arcs avec autant de sommets que l'argument qui lui est donné.
3. Écrire les fonctions `nouvelle_file`, `enfiler`, `est_vide` et `défiler` permettant d'implémenter la file d'attente à l'aide d'un tableau.
4. Écrire une fonction `calculer_chemin`(P, s, t) qui prend en argument la matrice d'adjacence du graphe H construit par la fonction `parcours_en_largeur`(G, s) et affiche le chemin dans H qui commence par le sommet source s et se termine avec le sommet t. Pour simplifier, on pourra afficher les sommets du chemin en ordre inverse : par exemple, si le chemin entre s et t est $s \to u \to v \to t$, on pourra afficher $\ll t\ v\ u\ s \gg$.