**Complexité et graphes**

Algorithmes de Dijsktra

1.1) voir le code de lecture du fichier pour construire la liste d’adjacence

2.1) Pour réaliser le parcours en largeur à partir d’une représentation matricielle on a besoin du code suivant :

def parcours\_Largeur(matrice):   
 visites = [] O(1)   
 for i in range(len(matrice)): n opérations  
 for j in range(len(matrice)): n opérations  
 if ((matrice[i][j] != 0) and (not j in visites) ) : O(1)  
 visites.append(j) O(1)

On peut donc évaluer la complexité de l’algorithme de parcours en largeur à partir d’une représentation matricielle.

Complexité : 1+ n \* (n\* (1 + 1))

En simplifiant cette explication, c’est-à-dire en négligeant les valeurs non significatives on peut simplifier notre complexité ce qui nous donne :

Complexité = n \* n = n²

La complexité du parcours en largeur d’un graphe représenté par une matrice d’adjacence est donc de n²

La fonction ci-dessus ne se trouve pas dans le code fourni

5.1)

Ligne 1 O(1) + O(n-1) = O(n) où n = nombres de sommets)

Ligne 2 : O(n)

Ligne 3 : O(1)

Ligne 4 - 5: O(1)

Ligne 6 : m itérations (m = nombres de voisins)

Ligne 7-8: O(1)

Ligne 4-8 : O(1) + m

Ligne 3-8 : n \* (O(1) + m) = O(n) + 2m = O(n+m)

Ligne 1-8 : O(n) + O(n) + O(n+m) = O(n+n+n+m) = O(n+m)

La complexité de la version « naïve » de l’algorithme de Dijkstra est donc en O(n+m)

5.2) Implémentation de la version « naïve » de l’algorithme de Dijkstra :

def dijkstra\_naif(self, origine: int):  
 distance = {sommet: inf for sommet in range(0, self.n)}  
 distance[origine] = 0  
  
 connus = [False] \* self.n  
  
 while not all(connus):  
 visite = None  
 for sommet in range(0, len(connus)):  
 if connus[sommet]:  
 continue  
 elif visite is None or distance[sommet] < distance[visite]:  
 visite = sommet  
  
 connus[visite] = True  
  
 for voisin, poids in self.liste\_adjacence[visite].items():  
 distanceMin = distance[visite] + poids  
 if distanceMin < distance[voisin]:  
 distance[voisin] = distanceMin  
  
 return distance

6.1) Implémentation de la version « naïve » de l’algorithme de Disjktra en utilisant un tableau contenant les candidats :

def dijkstra(self, origine: int):  
 distance = {sommet: inf for sommet in range(0, self.n)}  
 distance[origine] = 0  
  
 candidats = {sommet for sommet in range(0, self.n)}  
  
 while candidats:  
 visite = None  
 for sommet in range(0, len(candidats)):  
 if visite is None or distance[sommet] < distance[visite]:  
 visite = sommet  
  
 candidats.remove(visite)  
  
 for voisin, poids in self.liste\_adjacence[visite].items():  
 distanceMin = distance[visite] + poids  
 if distanceMin < distance[voisin]:  
 distance[voisin] = distanceMin  
 candidats.remove(voisin)  
  
 return distance

7.1)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 4 | 8 | 9 | 4 | 12 | 9 | 11 | 13 | 7 |

7.2)

Algorithme : insertion dans un tas

Données : tableau représentant le tas, clé à ajouter

Résultat : tableau représentant le tas après l’ajout

i = taille du tableau

ajout d’une case au tableau (pour pouvoir y ajouter la clé)

tant que ( i >1 && tableau [père de i] <= clé)

{

Tableau[i] = tableau[père de i] ;

i = père de i

}

Tableau [i] = v

7.3)

On ne tient pas compte des opérations non significatives pour la complexité (les opérations dont la complexité est de 0(1) comme les déclarations de variables. On se concentre ici sur la boucle *tant que*

On sait que l’algorithme s’arrête lorsque i/2k < 1.

On remplace ici *i* par *n.*

On peut donc en déduire que :

log2 (n/2k) <= log2 (1)

⬄ log2 (n) - log2 2k <=0 ⬄ log2 (n) – k log2 (2) ⬄k <= log2 (n)

7.4)

Dans le pire des cas il faut parcourir tout l’arbre afin que le nouvel objet soit bien placé. Sachant que la hauteur maximale d’un arbre contenant n éléments est *log2 n*, il faudra faire au maximum *log2 n* échanges.

7.5) def insertion\_tas(H, x):  
 cle = x[0] #type de x : couple -> (clé, valeur)  
 i = len(H)  
 H.append((None, None))  
 while (i // 2 > 1) and tableau[(i // 2)][0] >= cle:  
 tableau[(i)][0] = tableau[(i//2)][0]  
 i = 1//2  
 tableau[i] = x  
 return H

7.6) La complexité est identique à celle du pseudo code à la question 7.3). Elle est donc en O(log2n)

7.7)

def inserer(self, priorite, valeur):  
 element = (priorite, valeur)  
 idx = self.nMax = self.nMax + 1  
  
 self.tab.append(element)  
 while 0 < idx and self.tab[(idx - 1) // 2][0] > priorite:  
 self.tab[idx] = self.tab[(idx - 1) // 2]  
 self.index[(idx - 1) // 2] = idx  
 idx = (idx - 1) // 2  
  
 self.tab[idx] = element  
 self.index[valeur] = idx

7.8)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 13 | 4 | 8 | 9 | 4 | 12 | 9 | 11 |

7.9)

**Algorithme : retrait de la racine**

**Données : tableau représentant le tas (tab)**

**Résultat : tableau représentant le tas après le retrait de la racine (avec le dernier élément passé en première position, il n’est donc pas trié)**

racine = tab[0]

tab[0] = tab[dernier element]

supprimer (tab[dernier element])

retourner racine

7.10) On sait que la complexité de la suppression d’un élément d’un tableau (même trié) est en O(n).

On peut donc négliger la complexité des autre opérations (qui sont chacune en O(1)) et dire que la complexité de cet algorithme est en O(n)

7.11)

**Algorithme : retrait de la racine**

**Données : tableau représentant le tas (tab)**

**Résultat : tableau représentant le tas après le retrait de la racine et le tri**

racine = tab[0]

tab[0] = tab[dernier element]

objet = tableau[0] (objet = ancien dernier élément qu’on a placé en tête et qu’on doit replacer au bon endroit si il est supérieur à ses fils)

supprimer (tab[dernier element])

i = 0

tant que (2\*i + 1 < longueur du tableau {

j = 2\*i+1

si (j+1 < longueur du tableau et tableau[j+1] < tableau [j]

incrémenter j pour pouvoir continuer de parcourir le tableau

si objet <= tableau[j]

stopper la boucle

affecter tableau[j] à tableau[i] pour inverser le père et le fils

affecter j à i pour continuer le parcours

}

Tableau[i] = objet

Retourner la racine

*(à la fin de la fonction le tableau a perdu sa racine initiale et est de nouveau trié)*

7.12)

Comme pour l’algorithme d’insertion, le pire des cas serait celui ou il faudrait parcourir toute la hauteur de l’arbre. Comme on sait que la hauteur de l’arbre est de *log2 n,* la complexité de l’algorithme est en O(log2 n).

7.13)

def extraire\_racine(tableau):  
 racine = tableau[0]  
 tableau[0] = tableau[-1]  
 objet = tableau[0]  
 del(tableau[-1])  
 i=0  
 while 2\*i+1 < len(tableau):  
 j = 2\*i + 1  
 if (j+1 < len(tableau) and tableau[j+1][0] <tableau[j][0]):  
 j+=1  
 if objet[0] <= tableau[j][0]:  
 break  
 tableau[i] = tableau[j]  
 i = j  
 tableau[i] = objet  
 return racine

7.14)

def supprimer(self, valeurSupp):  
 idxSupp = self.index[valeurSupp]  
 element = (priorite, valeur) = self.tab[-1]  
  
 self.index[valeur] = idxSupp  
 self.index.pop(valeurSupp)  
  
 self.nMax -= 1  
 if self.nMax < 0:  
 return  
  
 idx = 0  
 while 2 \* idx + 1 < self.nMax:  
 jdx = 2 \* idx + 1  
  
 if jdx + 1 < self.nMax and self.tab[jdx + 1][0] < self.tab[jdx][0]:  
 jdx += 1  
  
 if element[0] <= self.tab[jdx][0]:  
 break  
  
 self.tab[idx] = self.tab[jdx]  
 self.index[self.tab[jdx][1]] = idx  
 idx = jdx  
  
 self.tab.pop()  
 self.tab[idx] = element  
 self.index[valeur] = idx

7.15)

Comme pour l’algorithme d’insertion et celui d’extraction de la racine, le pire des cas serait celui où il faudrait parcourir toute la hauteur de l’arbre. Comme cette hauteur vaut log2n et qu’on sait que la complexité des deux algorithmes cités précédemment est en O(log2n), celle de la suppression est équivalente.

Donc la complexité de l’algorithme de suppression est en O(log2n) (en sachant que la complexité pour retrouver l’indice d’un objet x dans un tableau est en O(1), on peut la négliger)

7.16) Le code implémenté est celui écrit à la question 7.14)

8.1)

Ligne 1) O(1)

Ligne 2) O(1)

Ligne 3) n-1 itérations

Ligne 4) O(1)

Ligne 5) n itérations

Ligne 6) complexité de INSERER -> O(log2 n)

Ligne 7) m itérations

Ligne 8) complexité de EXTRAIREMIN -> O(log2 n)

Ligne 9) O(1)

Ligne 10) p itérations

Ligne 11) O(1)

Ligne 12) Complexité de SUPPRIMER -> O(log2 n)

Ligne 13) O(1)

Ligne 14) complexité de INSERER -> O(log2 n)

Complexité = 1 + 1 + (n-1) \* 1 + n \* ( log2 n) + m \* (( log2 n) + 1 + p \* (1 + ( log2 n) + 1 + ( log2 n)))

Si on enlève les complexité non significatives en O(1) on obtient la complexité suivante :

Complexité = (n-1) + n \* ( log2 n) + m \* (( log2 n) + p \* (2 \* ( log2 n)))

Si on simplifie encore on obtient la complexité suivante :

n log2n + m \* log2n

= (n + m) log2n

La complexité de l’algorithme est donc en O((n + m) log2n)

8.2)

def dijkstra\_tas(self, origine: int):  
 distance = {sommet: inf for sommet in range(0, self.n)}  
 clefs = {sommet: inf for sommet in range(0, self.n)}  
 distance[origine] = 0  
 clefs[origine] = 0  
  
 tas = Tas()  
 tas.inserer(clefs[origine], origine)  
  
 while tas.nMax != -1:  
 distanceMin, visite = tas.extraireMin()  
 distance[visite] = clefs[visite]  
 for voisin, poids in self.liste\_adjacence[visite].items():  
 if clefs[voisin] > distance[visite] + poids:  
 clefs[voisin] = distance[visite] + poids  
 distance[voisin] = clefs[voisin]  
 tas.inserer(clefs[voisin], voisin)  
  
 return distance

9.1)

def majCle(self, objet, nouvellePriorite):  
 idx = self.index[objet]  
 self.tab[idx] = (nouvellePriorite, objet)  
  
 while 0 < idx and self.tab[(idx - 1) // 2][0] > self.tab[idx][0]:  
 save = self.tab[(idx - 1) // 2]  
  
 self.tab[(idx - 1) // 2] = self.tab[idx]  
 self.index[self.tab[idx][1]] = (idx - 1) // 2  
  
 self.tab[idx] = save  
 self.index[self.tab[(idx - 1) // 2][1]] = idx  
  
 idx = (idx - 1) // 2  
  
 save = self.tab[(idx - 1) // 2]

9.2)

Implémenter dijkstra en utilisant la majCle

10.1)

Faire les courbes de temps de calculs pour chaque graphe

10.2)

Qu’est-ce qui apporte le plus entre les modifications algorithmiques et les modifications d’implémentation ?