Rapport Algorithmique avancée

Partie 1 : TPs0-5

Camelia Mazouz
Groupe de TD : 03
N°étudiant : 22109872

Sommaire

1. Introduction

Ce document a pour but de détailler le code des Tps allant de 0 à 5.
Pour chaque TP vous trouverez une liste des fonctions réalisées ainsi que le rôle de chaque fonction.

Les fichiers .txt présents dans les répertoires, représentent les entrées claviers pour effectuer les tests.

Les fonctions libererMemoires servent à libérer la mémoire allouer pour chaque structure de données.

2. TP0 (3h)

Ce TP consiste à travailler sur des listes chainées et les arbres binaires.

Dans le répertoire TP0 vous trouverez :

1. Un fichier **arbres.h** qui contient les entêtes des fonctions liées aux arbres ainsi que la structure qui aide à implémenter un arbre.
2. Un fichier **listeschaineeh.h** qui contient les entêtes des fonctions liées aux listes chainées, ainsi que la structure d'une liste chainée et d'un point.
3. Un fichier **arbres.c** qui contient les définitions des fonctions suivantes :

a. Neoeud * nouvNoueud(char carac)
   - Qui créer et renvoie un nœud dont l'étiquette est le paramètre carac.

b. Nœud * rechercheNoeud(nœud *n, int numN) :
   - Prend en paramètre un arbre et un numero de nœud, et renvoie le nœud de l'arbre dont le numéro correspond au numéro de nœud passe en paramètre.

c. insererFG(nœud * n,int num,nœud *nvCel) :
   - Prend en parametre un arbre, le nouveau nœud à insérer entant que fils gauche sous le nœud numéro **num**.

d. insererFD(nœud * n,int num,nœud *nvCel) :
   - Prend en parametre un arbre, le nouveau nœud à insérer entant que fils droit sous le nœud numéro **num**.

e. Void parcoursPrefixe(nœud * n) :
   - Prend en paramètre un arbre et affiche le parcours préfixe de celui-ci.

f. Void parcoursInfixe(nœud * n) :
   - Prend en paramètre un arbre et affiche le parcours infixe de celui-ci.

g. Void parcoursSuffixe (nœud * n) :
   - Prend en paramètre un arbre et affiche le parcours suffixe de celui-ci.

4. Un fichier **listeChainees.c** qui contient les définitions des fonctions suivantes :

   a. Void afficher(cel * liste) :
      - Prend en parametre une liste, et affiche ses éléments.

   b. Cel * nouvCell(point p) :
      - Prend en paramètres un point et retourne une nouvelle cellule dont le point correspond au point passé en paramètre.

   c. Void insererCel(cel *liste,int pos,cel *cellule) :
      - Prend en parametre une liste et une cellule à insérer après la position **pos** dans la liste.

   d. Void suppCel(int pos,cel **liste) :
      - Prend en parametre l'adresse mémoire d'une liste et la position de la cellule à supprimer.

NB : Pour les listes chainées on considère que l'indexation des éléments commence à 1.

3. TP1 (2h)

Ce TP consiste à travailler sur les graphes enregistrés sous forme de matrices d'adjacences.

Dans le répertoire TP1 vous trouverez :

1. Un fichier **graphe.h** qui contient les entêtes des fonctions liées aux graphes ainsi que la structure qui aide à implémenter un graphe.
2. Un fichier **graphe.c** qui contient les contient les définitions des fonctions suivantes :

a. Graphe * chargeGraphe() :
   - Cette fonction retourne un graphe, demande à l'utilisateur de saisir le nombre de sommets d'un graphe ainsi que les arêtes et les stocks dans une matrice d'adjacence.

b. Void affiche(graphe * g) :
   - Cette fonction permet d'afficher la matrice d'adjacence d'un graphe passé en paramètre.

c. Void libererMemoire(graphe * g) :
   - Cette fonction permet de libérer la mémoire d'un graphe passé en paramètre.

d. Void AfficheMarques(int *tab,int taille,char *s) :
   - Cette fonction permet d'afficher les éléments d'un tableau passé en paramètre.

e. Void marquerVoisins(int **adjacence,int ordre,int s) :
   - Cette fonction permet de marquer tous les sommets du graphe passé en paramètre (adjacence), en partant du sommet s.

3.a Réponse aux questions :

1. Non, l'algorithme ne marque pas tous les sommets.
2. Pour faire en sorte que l'algorithme marque tous les sommets, il suffit de rajouter un counter (variable tousMarques) qu'on incrémente à chaque fois qu'on marque un sommet. Tant que cette variable est inférieure au nombre de sommets du graphe, on parcourt la liste des sommets marqués (variable marques) et on marque les voisins de tout sommet déjà marqué.
3. La complexité de calcul est $O(m^2)$
   a. Boucle do_while tourne n fois (n étant le nombre de sommets du graphe).
   b. La boucle for pour marquer les voisins tourne m fois aussi.

3.b Résultat d'exécution :

Exemple déroulé sur un graphe de 8 sommets et 11 arêtes.

```
Entrez les numeros des sommets (u,v) tq u et v sont voisins :
Arete n_1 :0 1

Arete n_2 :0 2

Arete n_3 :1 4

Arete n_4 :1 2

Arete n_5 :1 3

Arete n_6 :2 5

Arete n_7 :4 6

Arete n_8 :6 7

Arete n_9 :7 5

Arete n_10 :3 6

Arete n_11 :3 7

0 1 1 0 0 0 0 0
1 0 1 1 1 0 0 0
1 1 0 0 0 1 0 0
0 1 0 0 0 0 1 1
0 1 0 0 0 0 1 0
0 0 1 0 0 0 0 1
0 0 0 1 1 0 0 1
0 0 0 1 0 1 1 0

 ORDRE DES VOISINS MARQUES
 2  0  1  5  7  3  6  4
```

Figure 1

4. TP2 (2h)

Ce TP consiste à trouver le plus court chemin dans un graphe non pondéré avec File d'attente.

Dans le répertoire TP2 vous trouverez :

1. Un fichier **graphe.h** qui contient les entêtes des fonctions liées aux graphes ainsi que la structure qui aide à implémenter un graphe.
2. Un fichier **graphe.c** qui contient les contient les définitions des fonctions suivantes :

a. Graphe * chargeGraphe() :
   - Cette fonction retourne un graphe, demande à l'utilisateur de saisir le nombre de sommets d'un graphe ainsi que les arêtes et les stocks dans une matrice d'adjacence.

b. Void afficheGraphe(graphe * g) :
   - Cette fonction permet d'afficher la matrice d'adjacence d'un graphe passée en paramètre.

c. Void libererMemoire(graphe * g) :
   - Cette fonction permet de libérer la mémoire d'un graphe passé en paramètre.

d. Void afficheTab (int *tab,int taille,char *s) :
   - Cette fonction permet d'afficher les éléments d'un tableau passé en paramètre.

e. Void marquerVoisins(int **adjacence,int ordre,int s) :
   - Cette fonction permet de marquer tous les sommets du graphe passé en paramètre (adjacence), en partant du sommet s.

3. Un fichier **file.h** qui contient les entêtes des fonctions ainsi que, ainsi que la structure de données file.

4. Un fichier **file.c** qui contient les définitions des fonctions suivantes :

a. Void afficheFile(cel * liste) :
   - Prend en paramètre une file, et affiche ses éléments.

b. Cel * file() :
   - Retourne une file vide.

c. Cel * nouvCell(int u) :
   - Prend en paramètres un sommet u et retourne une nouvelle cellule dont le sommet correspond au sommet passé en paramètre.

d. Void enfiler(cel *liste,cel *cellule) :
   - Prend en paramètre une file et une cellule à insérer à la fin de la file.

e. int defile(cel **liste) :
   - Prend en paramètre l'adresse mémoire d'une file et supprime le premier élément de la file et le retourner.

f. Int estVide(cel *file) :
   - Prend en paramètre une file, retourne 1 si la file est vide, 0 sinon.

5. Un fichier **cheminGraphe.c** qui contient la fonction suivante :

a. Void plusCourtChemin(int **adjacence,int ordre, int s,int *l,int *pred) :
   - Cette fonction permet de trouver le plus court chemin entre le sommet **s** passé en paramètre et tous les autres sommets du graphe.

4.a Réponse aux questions :

1. L'algorithme utilisé est similaire au BFS et a une complexité de O(m).

4.b Résultat d'exécution :

Exemple déroulé sur un graphe de 8 sommets et 11 arêtes.

```
Entrez les numeros des sommets (u,v) tq u et v sont voisins :
Arete n_1 :0 1

Arete n_2 :0 2

Arete n_3 :1 4

Arete n_4 :1 2

Arete n_5 :1 3

Arete n_6 :2 5

Arete n_7 :4 6

Arete n_8 :6 7

Arete n_9 :7 5

Arete n_10 :3 6

Arete n_11 :3 7

0 1 1 0 0 0 0 0
1 0 1 1 1 0 0 0
1 1 0 0 0 1 0 0
0 1 0 0 0 0 1 1
0 1 0 0 0 0 1 0
0 0 1 0 0 0 0 1
0 0 0 1 1 0 0 1
0 0 0 1 0 1 1 0

PLUS COURT CHEMIN

FILE VIDE

 Liste des longueurs
 1  1  0  2  2  1  3  2
 Liste des predecesseurs
 1  1  0  2  2  1  3  2
```

Figure 2

4.c Activités supplémentaires :

1. Dans le cas d'un graphe connexe l'algorithme renvoie bine un arbre couvrant car il applique le même principe d'un algorithme BFS, dans le cas contraire l'algorithme nous renvoie l'arbre couvrant de la partie connexe qui contient le sommet **s**.

2. Le diamètre d'un graphe est la plus grande distance existante entre deux sommets d'un graphe :

$$diam(G) = max(d(u, v)|u, v \in V) = min(d|\forall u, v \in V , d(u, v) \leq d)$$

5. TP3 (2h)

Ce TP consiste à créer un arbre récursivement et le parcourir en préfixe, suffixe et infixe.

Dans le répertoire TP3 vous trouverez :

1. Un fichier **arbres.h** qui contient les entêtes des fonctions liées aux arbres ainsi que la structure qui aide à implémenter un arbre.
2. Un fichier **arbres.c** qui contient les fonctions suivantes :

a. Neoeud * nouvNoueud(char carac)
   - Qui créer et renvoie un nœud dont l'étiquette est le paramètre carac.

b. Nœud * rechercheNoeud(nœud *n, int numN) :
   - Prend en paramètre un arbre et un numero de nœud, et renvoie le nœud de l'arbre dont le numéro correspond au numéro de nœud passe en paramètre.

c. insererFG(nœud * n,int num,nœud *nvCel) :
   - Prend en parametre un arbre, le nouveau nœud à insérer entant que fils gauche sous le nœud numéro **num**.

d. insererFD(nœud * n,int num,nœud *nvCel) :
   - Prend en parametre un arbre, le nouveau nœud à insérer entant que fils droit sous le nœud numéro **num**.

e. Void parcoursPrefixe(nœud * n) :
   - Prend en paramètre un arbre et affiche le parcours préfixe de celui-ci.

f. Void parcoursInfixe(nœud * n) :
   - Prend en paramètre un arbre et affiche le parcours infixe de celui-ci.

g. Void parcoursSuffixe (nœud * n) :
   - Prend en paramètre un arbre et affiche le parcours suffixe de celui-ci.

h. Void arbreRecursif(nœud * racine) :
   - Cette fonction permet de créer un arbre récursivement en prenant en paramètre un nœud, demande à l'utilisateur si ce nœud a des enfants si oui :
     - Fils Gauche : on crée un nouveau nœud qu'on insère comme fils gauche au nœud père, puis on fait appel à la fonction sur ce nouveau nœud (le fils gauche).
     - FilsDroit : on crée un nouveau nœud qu'on insère comme fils Dorit au nœud père, puis on fait appel à la fonction sur ce nouveau nœud (le fils droit).

5.a Résultat d'exécution :

```
Est-ce que le noeud 'a' a des fils ? (0 : non/1 : oui) 1

Est-ce que le noeud 'a' a un fils gauche ? (0 : non/1 : oui) 1

Est-ce que le noeud 'b' a des fils ? (0 : non/1 : oui) 1

Est-ce que le noeud 'b' a un fils gauche ? (0 : non/1 : oui) 1

Est-ce que le noeud 'c' a des fils ? (0 : non/1 : oui) 0

Est-ce que le noeud 'b' a un fils droit ? (0 : non/1 : oui) 0

Est-ce que le noeud 'a' a un fils droit ? (0 : non/1 : oui) 1

Est-ce que le noeud 'd' a des fils ? (0 : non/1 : oui) 0

Pacrour prefixe : a  b  c  d
Pacrour infixe :  c  b  a  d
Pacrour suffixe :  c  b  d  a
```

Figure 3

6. TP4 (2h30)

Ce TP consiste à implémenter l'algorithme de Prim et l'essayer sur un graphe représenté par une matrice d'adjacence.

Dans le répertoire TP3 vous trouverez :

1. Un fichier **aretes.h** qui contient les entêtes des fonctions liées aux arrêtes ainsi que la structure qui aide à implémenter un arbre.
2. Un fichier **aretes.c** qui contient la fonction suivante :

   a. void* afficheArete(aretes * liste, int ordre)
      - Cette fonction prend en paramètre une liste d'arêtes et la taille de la liste et affiche toutes les arrêtes contenues dans la liste.

3. Un fichier **graphe.h** qui contient les entêtes des fonctions liées aux graphes ainsi que la structure qui aide à implémenter un graphe.

4. Un fichier **graphe.c** qui contient les contient les définitions des fonctions suivantes :

   a. Graphe * chargeGraphe() :
      - Cette fonction retourne un graphe, demande à l'utilisateur de saisir le nombre de sommets d'un graphe ainsi que les arêtes et les stocks dans une matrice d'adjacence.

   b. Void afficheGraphe(graphe * g) :
      - Cette fonction permet d'afficher la matrice d'adjacence d'un graphe passée en paramètre.

   c. Void libererMemoire(graphe * g) :
      - Cette fonction permet de libérer la mémoire d'un graphe passé en paramètre.

   d. Void afficheTab (int *tab,int taille,char *s) :
      - Cette fonction permet d'afficher les éléments d'un tableau passé en paramètre.

   e. Void marquerVoisins(int **adjacence,int ordre,int s) :
      - Cette fonction permet de marquer tous les sommets du graphe passé en paramètre (adjacence), en partant du sommet s.

5. Le fichier **prim.c** qui contient la fonction suivante :

   a. Aretes * prim(int **adjacence,int ordre, int s) :
      - Qui prend en paramètre une matrice d'adjacence, l'ordre du graphe et un sommet de départ s :
        - La fonction parcourt les sommets d'un graphe en partant du sommet s, et marque ses sommets voisins de poids minimal tout en enregistrant les arrêtes (sommet courant, sommet voisin poids minimal) dans une liste et la retourner.

6.a Résultat d'exécution :

Exemple déroulé sur un graphe de 8 sommets et 11 arêtes. (Fichier inout.txt)

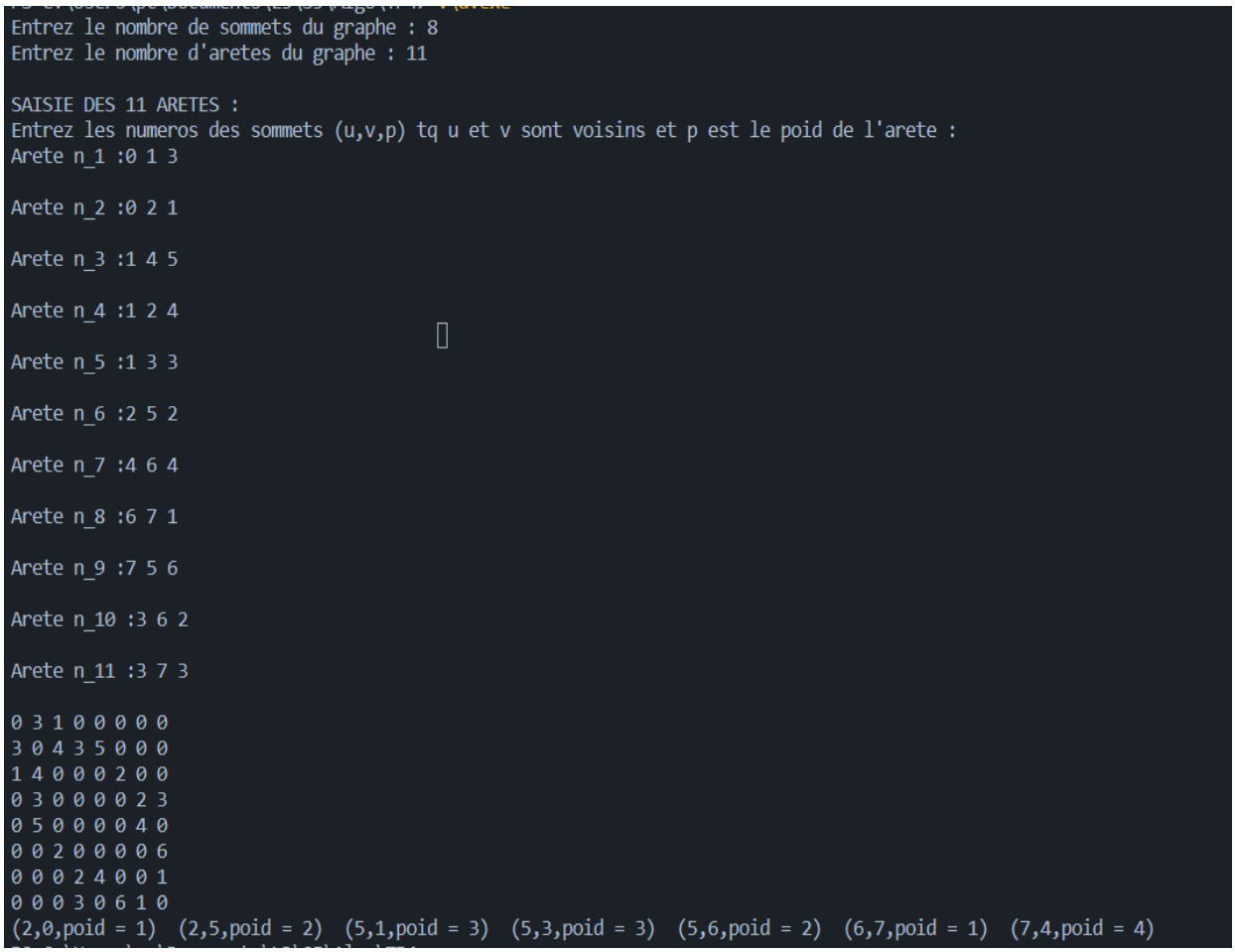

Figure 4

6.b Activités supplémentaires :

1. Le fichier **kruskal.c** qui contient ls fonctions suivantes :

   a. Aretes * chargeAretes(int ordre,int nbAretes) :
      - Cette fonction prend en paramètre l'ordre et la taille du graphe et demande à l'utilisateur de saisir les arêtes du graphe et les renvoie.

   b. Aretes * kruskal(graphe *graphe,int ordre,int s,int nint *nres):
      - Qui prend en paramètre une liste d'arêtes, l'ordre du graphe et son nombre d'arêtes **n**, un sommet de départ s et l'adresse de la variable **nres** pour retourner le nombre d'arêtes de l'arbre résultant :
        - La fonction retourne la liste d'arêtes de l'arbre couvrant du graphe **graphe** en partant du sommet **s**.

Exécution :

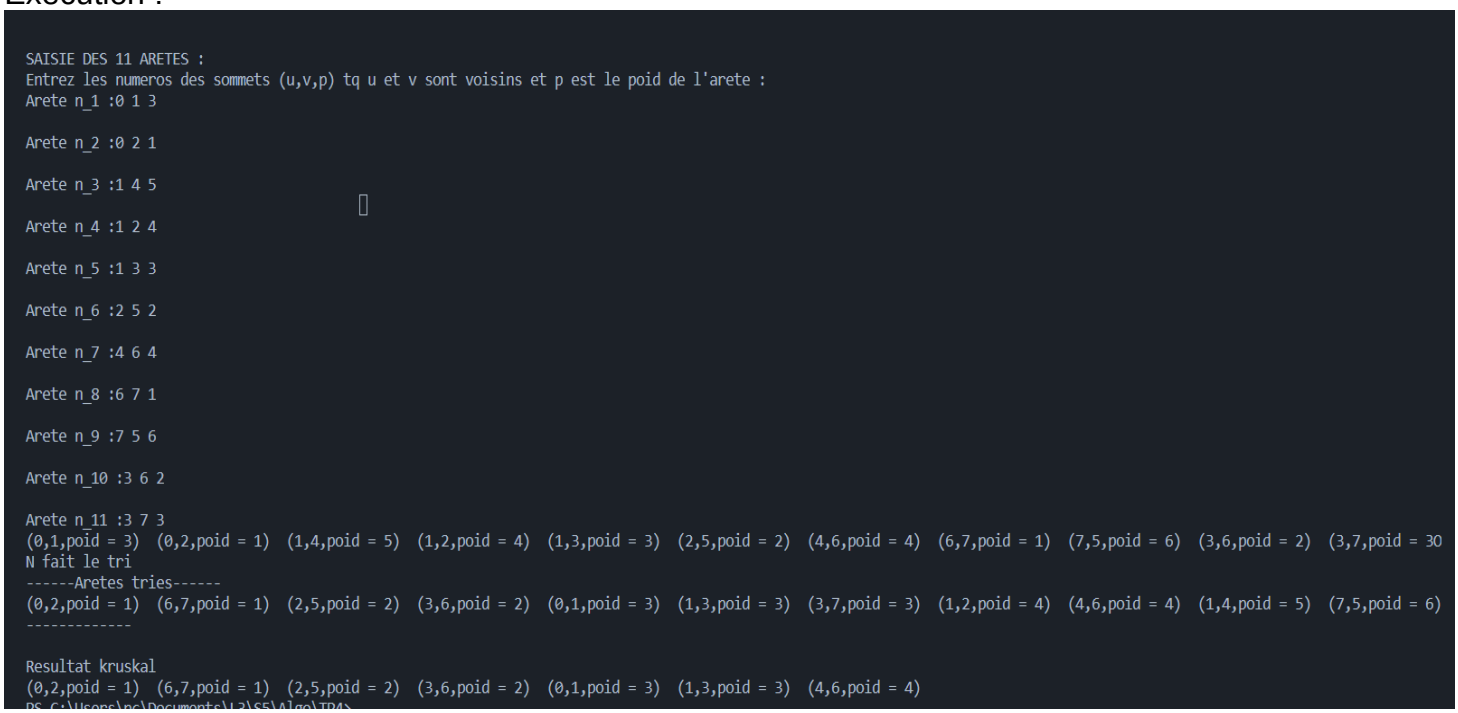

Figure 5

7. TP5 (2h)

Ce TP consiste à corriger un programme C qui calcule le nombre chromatique d'un graphe.

Le répertoire TP5 contient les fichiers suivants :

1. Un fichier **chatGPTGraphe.c** qui contient la structure d'un graphe et la fonction suivante :

a. Void addEdge(struct Graoh * graph,int u,int v) :
   - Qui prend en parametre un graphe, deux sommets voisins u et v et met à jour le graphe en précisant que u et v sont voisins.

b. Void welshPowell(struct Graph* graph) :
   - Cette fonction prend en paramètre un graphe.
   - Initialise les listes :
     - marques (pour mettre à jour les voisins marques).
     - degree (dans laquelle degree[i] = nombre de voisins du sommet i).
     - sorted_vertices (dans laquelle on mettra les sommets tries en ordre decroissant en fonction de leur degré).
     - Used_colors (dans la quelle on stock une couleur, si celle-ci est attribuée à un sommet **v** voisin du sommet qu'on veut colorer on met used_colors[color[v]] =1).
   - On tri les sommets par ordre décroissant en fonction de leur degré (avec un tri a bulle).
   - Pour chaque sommet de la liste **degree** on verifie la couleur de ses voisins et on met a jour le tableau used_colors
   - Si aucune couleur n'est disponible dans le tableau used_colors (toutes les valeurs sont a 1) on ajoute une nouvelle couleur et on incrémente le nombre de couleurs.
   - Sinon, on prend la première couleur non utilise et on l'attribue au sommet qu'on veut colorer.

7.a Résultat d'exécution :

Graphe de 5 sommets et 5 arrêtes : {(0,1),(0,,2),(1,2),(1,3),(2,3),(3,4)}

```
LISTE DES DEGRES EN ORDRE DECROISSANT
 1  2  3  0  4
Graph colored using 3 colors:
Vertex 0: Color 2
Vertex 1: Color 0
Vertex 2: Color 1
Vertex 3: Color 2
Vertex 4: Color 0
PS C:\Users\pc\Documents\L3\S5\Algo\TP5>
```

Figure 6

7.b Réponse aux questions :

1. Pour gérer les ensembles en JAVA il existe la classe HashSet
2. Pour gérer les dictionnaires en JAVA il existe l'interface Map et les classes :
   - **HashMap** (qui hérite de l'interface Map).
   - **LinkedHashMap** qui stocke les tuples (cles,valeurs) dans l'ordre d'insertion.
   - **TreeMap** qui permet de stocker les tuples (cle,valeur) dans un arbre binaire trie.
3. Pour gérer les files d'attentes en JAVA on peut utiliser l'interface Queue les classes :
   - **LinkedList** qui implémente l'interface Queue.
   - **ArrayDeque**
   - **ArrayList, Vector** qu'on peut aussi utiliser comme file avec la méthode add() pour enfiler et remove() pour défiler le premier élément.

4. Pour utiliser les piles en JAVA on peut utiliser la classe :
   - **ArrayDeque** qui implémente l'interface Deque
   - **ArrayList, Vector** qu'on peut aussi utiliser comme pile avec la méthode add() pour empiler et remove() pour dépiler le dernier élément.