TP17: Dijkstra: trajets dans Toulouse

Lecture de graphes orientés

On considère des graphes sans boucles pondérés, dont toutes les pondérations sont positives ou nulles. En C, nous utiliserons la structure suivante pour représenter les graphes par liste d'adjacence:

```c
typedef struct graph {
  int n; // nombre de sommets
  int n_max; // nombre maximal de sommets possible
  int m; // nombre d'arcs
  char** nom; // nom[i] donne le nom du sommet i
  int* degre; // degre[i] donne le degré du sommet i
  int** voisins; // voisins[i] est la liste d'adjacence de i
  float** poids; // poids[i][j] est le poids de l'arc (i, voisins[i][j])
} graph_t;
```

L'attribut `.m` ne sera pas utile dans le code, mais servira à la lecture et à l'écriture des graphes dans des fichiers textes.

Q1. Écrire une fonction `graph_t* graph_init(int n_max, int d)` créant un graphe vide en ayant réservé de la place pour n_{max} sommets, chacun de degré au plus d.

Q2. Écrire une fonction `void graph_add_vertex(graph_t* g, char* nom)` ajoutant au graphe un nouveau sommet nommé `nom`, initialement sans voisins. On supposera que le graphe a la place d'accueillir un nouveau sommet sans réallocation, et on se contentera de copier l'adresse de la chaîne de caractères `nom` sans en recopier le contenu.

Q3. Écrire une fonction `void graph_add_edge(graph_t* g, int u, int v, float w)` ajoutant au graphe `g` un arc entre les sommets d'indices u et v, de poids w.

Q4. Écrire une fonction `int graph_vertex_index(graph_t* g, char* nom)` prenant en entrée un graphe et un nom de sommet, et renvoyant l'indice i tel que `G->nom[i]` est le sommet en question. La fonction renverra -1 si le nom donné en argument n'est pas un sommet existant du graphe. *On pourra utiliser la fonction `strcmp` de la librairie `string.h`, dont n'oubliera pas qu'elle renvoie 0 en cas d'égalité.*

Q5. Dans une fonction `void test_construction()`, écrire du code permettant de créer le graphe G_0 représenté figure 2, en numérotant les sommets dans l'ordre alphabétique.

Q6. (Optionnelle) quelle structure de données pourrait-on ajouter au graphe pour améliorer la complexité de `graph_vertex_index` ?

Q7. Écrire une fonction `graph_free` prenant en entrée un graphe et libérant toute la mémoire allouée (y compris les noms des sommets).

On stocke un graphe $G = (S, A, w)$ dans un fichier texte au format suivant:

- Sur la première ligne, deux entiers n, m séparés d'une espace, donnant respectivement $|S|$ et $|A|$.
- Sur les n lignes suivantes, n chaînes de caractères **sans espace** donnant les noms des sommets. On suppose que les noms font moins de 50 caractères.
- Sur les m lignes suivantes, des triplets (u, v, d) avec $(u, v) \in A$ et $d = w(u, v)$.

Par exemple, la figure 2 montre le graphe G_0 et le contenu d'un fichier texte le représentant.

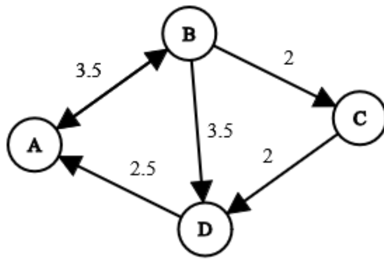

Figure 1: Graphe G_0

```
4 6
A
B
C
D
0 1 3.5
1 0 3.5
1 2 2.0
1 3 3.5
2 3 2.0
3 0 2.5
```

Figure 2: Graphe G_0 et le fichier `graphe0.txt` correspondant.

On peut supposer dans tous le TP que les graphes n'ont jamais de sommet de degré supérieur à 20.

Q8. Écrire une fonction `void graph_save(graph_t* g, char* filename)` prenant en entrée un graphe et un nom de fichier, et qui sauvegarde le graphe dans ce fichier, au format décrit plus haut.

Q9. Écrire une fonction `graph_t* graph_load(char* filename)` lisant un graphe au format décrit dans un fichier. On rappelle qu'on pourra supposer qu'aucun sommet n'est de degré strictement plus grand que 20.

Q10. Créer un fichier pour le graphe G_1 suivant, que l'on utilisera dans la suite pour tester l'algorithme de Dijkstra (attention, certains arcs sont bi-directionnels).

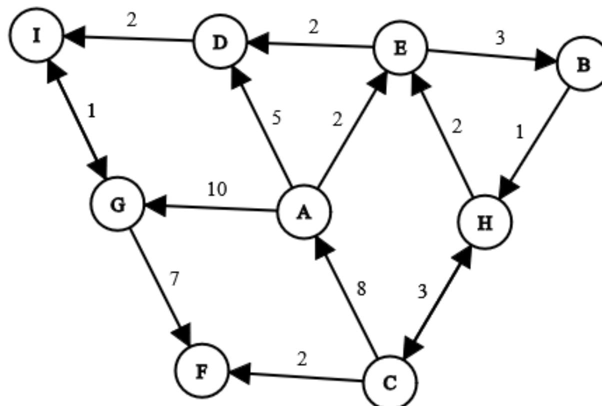

Figure 3: Graphe G_1

Chemins, algorithme de Dijkstra

Dans le code, on représente un chemin dans un graphe G par la suite des **indices** de ses sommets. Par exemple, dans G_0, le chemin C, D, A sera représenté par la suite d'indices $2, 3, 0$.

Q11. Écrire une fonction `bool check_path(graph_t* g, int* C, int p)` prenant en entrée un graphe g, un chemin C et un entier p donnant le nombre de sommets de C, et vérifiant que C est un chemin valide.

Rappelons le pseudo-code de l'algorithme de Dijkstra:

Algorithme 1 : PCC: Dijkstra

Entrée(s) : $G = (S, A, w)$ graphe pondéré avec $S = \{0, \dots, n-1\}$, $s \in S$
Sortie(s) : **Pred** tableau des prédecesseurs

1 **d** $\leftarrow$ tableau de taille n initialisé à $[+\infty, \dots, +\infty]$;
2 **Pred** $\leftarrow$ tableau de taille n initialisé à $[-1, \dots, -1]$;
3 **d**$[s] \leftarrow 0$;
4 **Pred**$[s] \leftarrow s$;
5 $Q \leftarrow$ ensemble contenant chaque élément de S;
6 **tant que** Q ***non vide*** **faire**
7 $\quad u \leftarrow$ extraire sommet de Q avec **d**$[u]$ minimal;
8 $\quad$ **pour** v ***voisin de*** u **faire**
9 $\quad\quad$ **si** ***d***$[u] + w(u, v) <$ ***d***$[v]$ **alors**
10 $\quad\quad\quad$ **d**$[v] \leftarrow$ **d**$[u] + w(u, v)$;
11 $\quad\quad\quad$ **Pred**$[v] \leftarrow u$;
12 **retourner** ***Pred***

En C, on utilisera un tableau de booléens `bool* Q` pour représenter l'ensemble des sommets restants à traiter, ainsi qu'un entier `int taille_Q` donnant le nombre de sommets.
En C, la librairie `<math.h>` a une constante `INFINITY` de type `float` représentant $+\infty$.

Q12. Écrire une fonction `int extraire_min(bool* Q, float* d, int n)` prenant en entrée deux tableaux Q et d, et renvoyant un indice i tel que $Q[i] = 1$ et pour lequel $d[i]$ est minimal.

Q13. Implémenter l'algorithme de Dijkstra:

```
1 /* Renvoie le tableau des prédécesseurs de l'arborescence
2    de l'algorithme de Dijkstra lancé dans g depuis s. */
3 int* dijkstra_source(graph_t* g, int s);
```

Q14. Tester la fonction `dijkstra_source` sur le graphe G_1, vérifiant bien les résultats obtenus.

Toulouse

Le fichier `graphe_toulouse.txt` représente le graphe des rues de Toulouse dans une zone de 1.5km autour de la DE02. Il comporte $n = 54837$ sommets et $m = 125146$ arcs.

Voici plusieurs points d'intérêt de la carte, et les **noms** (pas les indices !) des sommets correspondant:

- Couvent des Jacobins: 9108344823
- Square Boulingrin: 245534479_1419908431_1
- Pavillon de thé du jardin Pierre Baudis: 9759123700

- Tombe de Federica Montseny: 1359276666_1359276696_2
- Jardin Edmond Michelet: 314411065_12796208682_2
- Théâtre Garonne: 246542808_303770407_1
- Place Saint-Pierre: 1378020951

L'archive du TP contient un fichier Python `toulouse.py` avec trois modes d'exécution. Le lancer sans aucun argument (`python3 toulouse.py`) affiche le graphe, en bleu. Vous pouvez aussi le lancer en précisant deux fichiers `arbo.txt` et `chemin.txt` en argument. Le fichier `arbo.txt` doit contenir une arborescence de parcours, en mettant sur chaque ligne le nom d'un sommet et le nom de son prédécesseur. Le fichier `chemin.txt` doit contenir un nom de sommet par ligne. Le programme trace alors le graphe en bleu, l'arborescence de parcours en jaune, et le chemin en orange.

Q15. Tester le programme `toulouse.py` sans arguments, puis en utilisant les fichiers `arbo_test.txt` et `chemin_test.txt` de l'archive. Attention: pour que le programme Python fonctionne correctement, les fichiers `graphe_toulouse.txt` et `positions_toulouse.txt` doivent être placés dans le même dossier que `toulouse.py`.

Q16. Écrire une fonction `int* chemin(int* pred, int s, int t, int n, int* longueur)` prenant en entrée un tableau de prédécesseurs renvoyé par l'algorithme de Dijkstra, le sommet s source de l'arborescence, ainsi qu'un sommet t, et qui renvoie le chemin de s à t donné par l'arborescence. L'entier n donnera le nombre total de sommets du graphe, et on pourra l'utiliser pour allouer le chemin. La fonction stockera dans `*longueur` le nombre de sommets du chemin.

Q17. Modifier le main pour que le programme prenne en entrée un nom de fichier graphe, deux **noms** de sommets, et deux noms de fichier arborescence et chemin, et calcule un plus court chemin entre les deux sommets, en stockant l'arborescence et le chemin généré dans les fichiers spécifiés:

```
fredfrigo@ordi~$ ./a.out graphe_1.txt A D arbo.txt chemin.txt
Plus court chemin de A à D calculé.
Arborescence stockée dans arbo.txt
Chemin stocké dans chemin.txt
```

Attention, les fichiers générés doivent contenir les **noms** des sommets, et pas leurs indices.

Q18. Testez votre programme en traçant le plus court chemin des Jacobins jusqu'à la place Saint-Pierre. Combien de temps le programme C prend-il environ ?

Q19. Recompilez avec l'option `-O3` qui active les optimisations de GCC et testez à nouveau.

Lorsque l'on utilise l'algorithme de Dijkstra pour calculer le plus court chemin d'un point à un autre, il n'y a pas besoin de calculer toute l'arborescence: il suffit de s'arrêter une fois que le sommet cible est extrait de Q. Le temps de calcul dépendra alors de la distance entre les deux sommets: plus ils seront proche, et plus vite le sommet cible sera traité par l'algorithme.

Q20. Écrire une fonction `int* dijkstra_target(graph_t* g, int s, int t)` appliquant l'algorithme de Dijkstra en s'arrêtant dès que le sommet t est extrait. Attention aux fuites mémoires !

Q21. Modifiez votre programme pour qu'il utilise `dijkstra_target` plutôt que `dijkstra_source`. Vérifiez que le trajet des Jacobins jusqu'à Saint-Pierre est calculé en moins de temps, et que l'arborescence générée est bien plus réduite.

Q22. Testez un trajet plus long, par exemple du Boulingrin jusqu'au jardin Pierre Baudis.

Tas binaire

L'algorithme de Dijkstra, tel qu'implémenté pour le moment, a une complexité $\mathcal{O}(n^2)$, car l'extraction du sommet de Q prend un temps $\mathcal{O}(n)$. Nous avons vu en cours qu'utiliser une file de priorité implémentée par tas binaire permet d'atteindre une complexité en $\mathcal{O}((n+m)\log n)$.

Q23. Pour le graphe de Toulouse du TP, que vaut n^2 ? et $(n+m)\log_2 n$?

Nous allons implémenter des tas binaires pouvant stocker des éléments tous distincts et compris entre 0 et $n-1$. Les tas implémentés plus tôt dans l'année permettent l'ajout et l'extraction du minimum en temps logarithmique, mais:

- L'ordre utilisé était l'ordre naturel $<$, alors que dans l'algorithme de Dijkstra, on veut comparer deux sommets u, v selon les valeurs de $d[u]$ et $d[v]$.
- On ne peut pas facilement **modifier** la priorité d'un élément, car on n'a aucun moyen de savoir où se trouve un élément u donné dans le tas.

Afin de résoudre le premier problème, on ajoute simplement au tas un pointeur vers le tableau d. Pour le second problème, on ajoute un attribut `.rang` tel que `.rang[u]` donne le rang dans le tas de l'élément u, ou -1 si l'élément n'est pas présent. Par exemple, si le tas a de la place pour $n = 10$ éléments, et contient pour l'instant:

$$[7, 3, 8, 5, 0, 2]$$

alors le tableau des rangs contiendra:

$$[4, -1, 5, 1, -1, 3, -1, 0, -1, -1]$$

car 0 est à la position **4**, 1 est absent, 2 est à la position 5, etc...

On utilisera donc l'interface suivante :

```c
typedef struct {
  int* T; // tableau des éléments
  float* d; // valeurs à utiliser pour comparer les éléments
  int* R; // tableau des rangs
  int n; // nombre actuel d'éléments
  int n_max ; // nombre maximal d'éléments autorisés
} heap_t;

/* Renvoie un tas vide ayant la place pour n éléments. */
heap_t* heap_init(int n_max);

/* Renvoie true si h est vide, false sinon */
bool heap_is_empty(heap_t* h);

/* Ajoute u dans h avec la priorité w. Modifie la priorité de u
   si u était déjà dans h.
   Précondition: la priorité ne peut qu'être améliorée: si w0 est la
   priorité actuelle de u, alors w <= w0. */
void heap_add(heap_t* h, int u, float w);

/* Renvoie l'élément de priorité maximale de h, c'est à dire
   celui pour lequel .d est minimal. Supprime l'élément renvoyé. */
int heap_extract(heap_t* h);
```

Q24. Recopier l'interface dans un fichier header (en n'oubliant pas les *include guards*).

La seule différence par rapport aux tas implémentés il y a quelques TP est que l'on doit garder une trace des rangs des éléments, afin de savoir dans quelle case écrire pour modifier la priorité.

Q25. Écrire une fonction `void swap(heap_t* h, int i, int j)` échangeant les éléments de rangs i et j de h, en prenant garde à modifier le tableau des rangs en conséquence. Par exemple, si le tas est de taille maximale 5 et contient:

$$[1, 3, 4]$$

avec comme tableau des rangs:

$$[-1, 0, -1, 1, 2]$$

alors après avoir échangé les éléments de rang 0 et 2, le tas contiendra:

$$[4, 3, 1]$$

et le tableau des rangs contiendra:

$$[-1, 2, -1, 1, 0]$$

Q26. Implémenter toutes les opérations des tas.

Q27. Écrire une nouvelle version de l'algorithme de Dijkstra utilisant cette structure, et vérifier que le temps d'exécution est bien plus court.