Projet IPI : chemins de poids optimum.

2017-2018

Le projet du cours de programmation impérative consiste en le rendu d'un code en C. Vous devez faire ce projet seul et rendre votre code sur la plateforme `http://exam.ensiie.fr` sur le dépôt `Projet-IPI-2017` avant le 15 janvier 2018.

Votre code devra contenir un fichier C par exercice et un (petit) rapport. Ce sujet comporte 3 exercices. Le fichier du premier exercice sera nommé `exo1.c`, celui du deuxième exercice `exo2.c`, celui du troisième `exo3.c` et le rapport sera nommé `rapport.pdf`. Tous ces fichiers devront être archivés en un fichier `nom-prenom.tar.gz` (où `nom` et `prenom` devront être, bien entendu, remplacés par votre nom et votre prénom) avec la commande suivante

```
tar -cvzf nom-prenom.tar.gz exo*.c rapport.pdf
```

Avec ce sujet vous est fourni, dans le dossier `/pub/IPI/2017-2018/` auquel vous pouvez accéder depuis les machines de l'école, un dossier compressé `projet.tar.gz` que vous pouvez copier et décompresser avec la commande.

```
tar -xvzf projet.tar.gz
```

Ce dossier contient:

- Des instances de test pour chaque exercice pour vous permettre de tester vos codes. Il y a des 10 tests unitaires pour les exercices 1 et 2 et 18 pour le 3, présents dans les dossiers `tests/exo1`, `tests/exo2` et `tests/exo3`. Pour chaque test des exercices 1 et 2, il y a un dossier (numéroté de 01 à 10), contenant deux fichiers, `input` et `output`, le premier contenant l'instance et le second la sortie attendue. Pour l'exercice 3, seul un fichier `input` vous est donné.

- Des fichiers `exo1.c`, `exo2.c` et `exo3.c` que vous devez compléter. Ces fichiers sont déjà préécrits pour lire l'entrée.

- Un script bash par exercice, `exec1.sh`, `exec2.sh`, et `exec3.sh` pour lancer ces tests sur vos codes. Chaque scripts compile votre code (et vous renvoie les erreurs ou les warnings s'il y en a), le teste, s'arrête si votre code calcule plus de 3 secondes et compare votre sortie à celle qui est attendue. Voici quelques exemples d'utilisation des scripts:

- Lancer un script sans argument lance tous les tests : `./exec1.sh` teste votre code sur les 10 instances de l'exercice 1, vous indique pour chaque test s'il est réussi et pourquoi puis un résumé vous disant combien de tests ont été réussis
- Langer un script avec un argument entre 01, 02, ... et 10 lance le test correspondant à l'argument : `./exec1.sh 03` teste votre code de l'exercice 1 sur l'instance du dossier `test/exo1/03` et vous indique s'il est réussi et pourquoi.
- Langer un script avec plusieurs argument entre 01, 02, ... et 10 lance tous les tests correspondant aux arguments : `./exec1.sh 02 04 07` teste votre code de l'exercice 1 sur les instances des dossiers `test/exo1/02`, `test/exo1/04` et `test/exo1/07`, vous indique pour chaque test s'il est réussi et pourquoi puis un résumé vous disant combien de tests ont été réussis.
- Vous pouvez utiliser l'option -q. Dans ce cas, seul le résumé vous est donné : `./exec1.sh 02 04 07 -q` fait la même chose que précédemment mais seul le résumé est affiché.
- `./exec3.sh` propose une petite particularité qui sera expliquée dans la partie concernée du sujet.

Votre code sera testé sur des instances similaires à celles des tests fournis, pas nécessairement les mêmes. (Il n'est pas nécessaire de passer tous les tests pour avoir des points.)

Consignes importantes :

- Le fichier compressé que vous devez renvoyer à la fin ne doit contenir que les fichiers exo1.c, exo2.c, exo3.c et rapport.pdf, et **en aucun cas** d'autres fichiers (ni tests, ni script, ni autre chose).
- Votre rapport est limité à 8 pages, page de garde et images comprises. Il est inutile de tout décrire dans le rapport. Uniquement ce qui est intéressant. En particulier, il est inutile de mettre des extraits ou des captures d'écran du code, ou encore de lister toutes vos fonctions une par une!
- Vos code doivent tous compiler sans erreur ni warning avec les options `-Wall -Wextra` de `gcc`.

Le non respect de ces consignes entraînera automatiquement la note de **0/20** à votre projet, sans tenir compte ni de sa qualité ni du travail que vous avez fourni pour le produire!

1 Présentation du sujet

Le sujet consiste en la recherche de chemins de poids minimum dans les graphes orientés entre un nœud s et un nœud t. Les trois algorithmes présentés ici sont l'algorithme de parcours en largeur, l'algorithme de Dijkstra et l'algorithme A*, le second pouvant être vu comme une généralisation du premier et le troisième comme une généralisation du second.

1.1 Parcours en largeur

L'algorithme de parcours en largeur trouve dans un graphe orienté $G = (V, A)$ un chemin avec un nombre minimum d'arcs entre un nœud s et un nœud t, autrement dit un chemin de poids minimum où les poids de tous les arcs sont de poids 1.

L'idée est d'énumérer la liste V_1 des successeurs de s, puis la liste V_2 des successeurs des nœuds de V_1 qui ne sont ni s ni dans V_1, puis la liste V_3 des successeurs des nœuds de V_2 qui ne sont ni s ni dans V_1 ni dans V_2, ... L'ensemble V_i est l'ensemble des nœuds à distance i de s. Lorsqu'un ensemble V_d contient t, on a trouvé la distance d entre s et t. Si, à une itération $V_i = \emptyset$ et si on a pas trouvé t alors il n'existe pas de chemin de s à t.

La Figure 1 donne un exemple de graphe avec les ensembles V_0 à V_3, V_0 étant l'ensemble réduit à s.

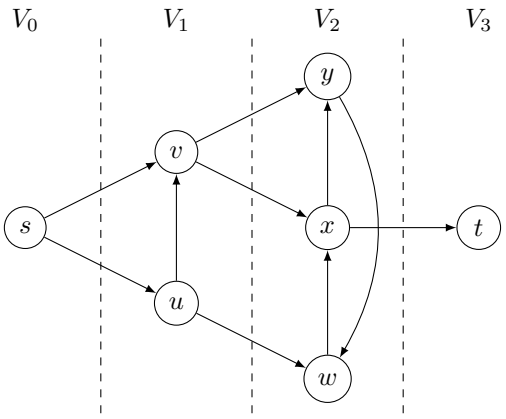

Figure 1: Exemple de graphe orienté où on a représenté les ensembles V_0, V_1, V_2 et V_3.

Pour reconstruire un chemin de s à t de taille minimum, il existe plusieurs méthodes, en voici deux:

- si t est dans V_d c'est qu'il existe un nœud v de V_{d-1} et un arc de v à t. Si ce nœud est dans V_{d-1} c'est qu'il existe un nœud $v' \in V_{d-2}$ et un arc de v' à v ... On peut ainsi énumérer des prédecesseurs de proche en proche jusqu'à arriver à s (qui constitue l'ensemble V_0).

- on maintient au fur et à mesure une liste p de prédécesseurs qui associe à chaque nœud v un prédécesseur de v dans un plus court chemin de s à v. Lorsque v est ajouté à V_i, on définit par $p(v)$ le nœud de V_{i-1}, nécessairement prédecesseur de v, qui a permis d'ajouter v à V_i.

1.2 Algorithme de Dijkstra

L'algorithme de Dijkstra permet dans un graphe orienté $G = (V, A)$ où les arcs sont valués avec des poids positifs $\omega : A \to \mathbb{R}^+$, de trouver un chemin de poids minimum entre deux nœuds s et t, c'est-à-dire un chemin dont la somme des poids des arcs est minimum.

Cet algorithme peut être vu comme une généralisation du parcours en largeur dans le sens où, comme ce dernier, il énumère les nœuds par ordre croissant de distance depuis s jusqu'à trouver t. De plus, si tous les poids des graphe sont des entiers, alors remplacer dans G chaque arc (u, v) de poids p par un chemin $(u, u_1, u_2, \ldots, u_{p-1}, v)$ (en ajoutant ainsi $p-1$ nœuds et arcs), alors l'algorithme de Dijkstra agit exactement comme l'algorithme de parcours en largeur. Il est déconseillé de coder l'algorithme de Dijkstra ainsi même quand les poids sont des entiers, car les poids peuvent être très élevés.

Dans le cas général, voici le pseudo code de l'algorithme :

Algorithme 1 Algorithme de Dijkstra

Entrées: Un graphe orienté $G = (V, A)$ orienté, deux nœuds s et t, et une fonction de poids $\omega : A \to \mathbb{R}^+$.

Sorties: Le poids d'un chemin P de poids minimum reliant s à t ou $+\infty$ si un tel chemin n'existe pas.

```
1:  d(s) = 0
2:  Pour v ∈ V Faire
3:      d(v) = +∞
4:  Tant que True Faire
5:      u ← un nœud non visité minimisant d(u)
6:      Marquer u comme visité
7:      Si u = t Alors
8:          Renvoyer d(t)
9:      Pour chaque successeur v de u Faire
10:         d(v) ← min(d(v), d(u) + ω(u, v))
```

Pour reconstruire un chemin de s à t de poids minimum, il existe plusieurs méthodes, en voici deux:

- si la distance de s à t est $d(t)$ c'est qu'il existe un nœud v et un arc de v à t de poids $d(t) - d(v)$. Si ce nœud est de distance $d(v)$, c'est qu'il existe un nœud v' et un arc de v' à v de poids $d(v) - d(v')$, ... On peut ainsi énumérer des prédecesseurs de proche en proche jusqu'à arriver à s (de distance $d(s) = 0$).

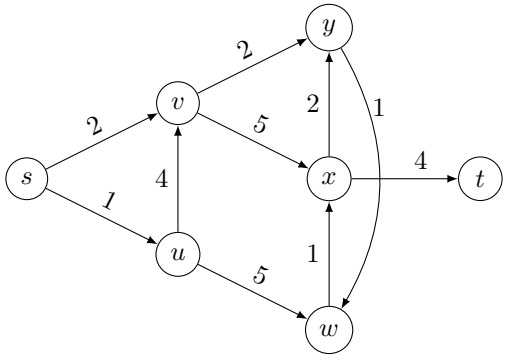

Figure 2: Exemple de graphe orienté. Les distances d de s vers les autres nœuds sont : $d(s) = 0, d(u) = 1 d(v) = 2, d(y) = 4, d(w) = 5, d(x) = 6, d(t) = 10$. L'algorithme de Dijkstra visite tous ces nœuds dans cet ordre.

- on maintient au fur et à mesure une liste p de prédécesseurs qui associe à chaque nœud v un prédécesseur de v dans un chemin de poids minimum de s à v. Lorsque $d(v)$ est mis à jour, c'est-à-dire qu'on trouve un nœud u tel que $d(u) + \omega(u, v) < d(v)$, c'est-à-dire que, jusqu'à présent, le meilleur moyen d'accéder à v depuis s est de passer par u, alors on met aussi à jour $p(v)$ avec u.

2 Exercices

Dans cette partie, chacun des exercices suivants consiste à coder un programme en c qui calcule un chemin de poids minimum entre deux points.

Vos fonctions n'auront aucun paramètre en entrée. Celle-ci sera fournie dans l'entrée standard avec la fonction `scanf` ou la fonction `fgets`. De même, vos fonctions n'auront pas de valeur de retour. Celle-ci sera écrite directement dans la sortie standard avec la fonction `printf`. Les fichiers `exo1.c`, `exo2.c` et `exo3.c` qui vous sont fournis contiennent déjà un main capable de lire les entrées. Vous n'avez normalement pas à vous focaliser sur ce point. Cependant vous pouvez très bien modifier ce code si vous le souhaiter.

Vos fichiers peuvent se contenter d'une fonction `main`, mais il vous est possible d'ajouter autant de fonctions que vous le souhaiter dans votre fichier. Chacun de vos fichiers devra se suffire à lui-même (pas d'appel à d'autres fichiers, excepté des bibliothèques standards).

2.1 Exercice 1

Implantez dans un fichier nommé `exo1.c` l'algorithme de parcours en largeur.

Vous devez, dans votre rapport expliquer quels sont les choix que vous avez fait pour implanter cet algorithme. En particulier de quelles structures de données (tableau, listes, files, ...) avez vous eu besoin et comment les avez-vous utilisées?

2.1.1 Description des entrées

Le programme sera lancé avec, en entrée standard, la description d'un graphe orienté $G = (V, A)$ et deux nœuds s et t de V.

- La première ligne sera un entier n, correspondant au nombre de nœuds du graphe G, numérotés de 1 à n.
- Les n lignes suivantes seront une suite de n entiers égaux à 0 ou 1, séparés par des espaces. Le j^e nombre de la i^e ligne vaut 1 s'il existe un arc dans A entre le nœud i et le nœud j et 0 sinon.
- La dernière ligne contient deux entiers entre 1 et n séparés par une espace et correspondent respectivement aux numéros des nœuds s et t.

On garanti que, quelque soit l'instance, s'il existe un chemin de poids minimum de s à t alors ce chemin est unique. Les graphes ont entre 5 et 50 nœuds.

2.1.2 Description des sorties

Votre programme doit afficher dans la sortie standard le chemin de taille minimum entre s et t si celui ci existe. Dans ce cas, si le chemin trouvé est

$(v_1 = s, v_2, v_3, \ldots, v_{p-1}, v_p = t)$, alors votre programme doit afficher p lignes où la ligne i contient le numéro du nœud v_i.

Dans le cas où ce chemin n'existe pas, vous devez afficher la chaîne de caractères `Not connected`.

Par exemple, dans le graphe de la Figure 1. Vous recevez en entrée les lignes suivantes (les nœuds sont numérotés dans cet ordre : s, u, v, w, x, y, t).

Listing 1: Exemple d'entree

```
7
0 1 1 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 1 0
0 0 0 0 1 0 0
0 0 0 0 0 1 1
0 0 0 1 0 0 0
0 0 0 0 0 0 0
1 7
```

On attend la sortie suivante :

Listing 2: Exemple de sortie

```
1
3
5
7
```

2.1.3 Tester votre programme

2.2 Exercice 2

Implantez dans un fichier nommé `exo2.c` l'algorithme de Dijkstra.

Vous devez, dans votre rapport expliquer quels sont les choix que vous avez fait pour implanter cet algorithme. En particulier de quelles structures de données (tableau, listes, files, ...) avez vous eu besoin et comment les avez-vous utilisées?

2.2.1 Description des entrées

Le programme sera lancé avec, en entrée standard, la description d'un graphe orienté $G = (V, A)$, deux nœuds s et t de V, et les poids ω de chaque arc.

- La première ligne sera un entier n, correspondant au nombre de nœuds du graphe G, numérotés de 1 à n.
- Les n lignes suivantes seront une suite de n entiers supérieur ou égaux à -1, séparés par des espaces. Le j^e nombre de la i^e ligne vaut -1 s'il n'existe pas d'arc dans A entre le nœud i et le nœud j. Si cet arc existe alors l'entier sera le poids $\omega(i, j)$.

- La dernière ligne contient deux entiers entre 1 et n séparés par une espace et correspondent respectivement aux numéros des nœuds s et t.

On garanti que, quelque soit l'instance, s'il existe un chemin de poids minimum de s à t alors ce chemin est unique. Les graphes ont entre 5 et 50 nœuds, les poids varies de 1 à 1000000000.

2.2.2 Description des sorties

Votre programme doit afficher dans la sortie standard le chemin de poids minimum entre s et t si celui ci existe. Dans ce cas, si le chemin trouvé est $(v_1 = s, v_2, v_3, \ldots, v_{p-1}, v_p = t)$, alors votre programme doit afficher p lignes où la ligne i contient le numéro du nœud v_i.

Dans le cas où ce chemin n'existe pas, vous devez afficher la chaîne de caractères `Not connected`.

Par exemple, dans le graphe de la Figure 2. Vous recevez en entrée les lignes suivantes (les nœuds sont numérotés dans cet ordre : s, u, v, w, x, y, t).

Listing 3: Exemple d'entree

```
7
−1  1  2 −1 −1 −1 −1
−1 −1  4  5 −1 −1 −1
−1 −1 −1 −1  5  2 −1
−1 −1 −1 −1  1 −1 −1
−1 −1 −1 −1 −1  2  4
−1 −1 −1  1 −1 −1 −1
−1 −1 −1 −1 −1 −1 −1
1 7
```

On attend la sortie suivante :

Listing 4: Exemple de sortie

```
1
3
6
4
5
7
```

2.3 Exercice 3

Dans ce dernier exercice, ce n'est pas un graphe que vous recevez en entrée mais la carte d'un labyrinthe ainsi qu'une durée. Vous devez indiquer comment aller de l'entrée à la sortie avant que la durée ne soit écoulée.

Vous devez, dans votre rapport expliquer quels sont les choix que vous avez fait pour implanter vos algorithmes. En particulier de quelles structures de

données (tableau, listes, files, ...) avez vous eu besoin et comment les avez-vous utilisées?

2.3.1 Description des entrées

Le programme sera lancé avec, en entrée standard, la description d'un labyrinthe et des durées.

- La première ligne sera un entier n
- Les n lignes suivantes seront une chaîne de n caractères. Chacun des caractère représente un espace, un mur ou un objet dans le labyrinthe.
  - Un point : `.`, représente un espace vide. On peut se déplacer d'un espace vide à un autre.
  - `E`, représente le point de départ. Elle est toujours en haut à gauche du labyrinthe.
  - `S`, représente la destination. Elle est toujours en bas à droite du labyrinthe.
  - `X`, représente un mur. Il n'est pas possible de se déplacer sur un mur.
  - `a` et `A`, représentent respectivement une clef et une porte. Il faut se déplacer sur la clef pour la ramasser. Il n'est possible de se déplacer sur une porte que si on dispose de la clef, auquel cas, elle agit comme un espace vide.
  - Les autres symboles : `*`, `%`, `$`, `#`, `&`, `+`, `-`, `@`, `^`, `£`, représentent des téléporteurs. Ils vont toujours par paires des mêmes symboles. Lorsqu'on se déplace sur un symbole de téléporteur, il est possible de se téléporter instantanément sur l'autre téléporteur ayant le même symbole.
- La dernière ligne contient un entier, le nombre de seconde avant lequel vous devez avoir rejoint la sortie. Se déplacer d'un cran dans le labyrinthe prend une seconde par contre se téléporter ou ramasser une clef est instantané. Le compteur n'est pas strict, vous pouvez parfois arriver avant la fin. Vous pouvez aussi arriver quand le compteur atteint 0.

On garanti que, quelque soit l'instance, il existe un chemin de l'entrée à la sortie du labyrinthe. La taille du labyrinthe varie de 5 à 1000. Le compteur est toujours supérieur ou égal à la durée d'un plus court chemin de E à S.

2.3.2 Description des sorties

Votre programme doit donner en sortie la liste des actions à effectuer pour sortir du labyrinthe dans les temps impartis. Il y a 5 actions possibles: `HAUT`, `BAS`,

`GAUCHE`, `DROITE` et `TP`. Les 4 premières actions indiquent un déplacement qui vous prend une seconde (le labyrinthe vous est décrit de haut en bas et de gauche à droite) et la dernière, si on est sur un téléporteur, indique de se téléporter, ce qui est instantané.

Par exemple, si vous recevez l'entrée suivante :

Listing 5: Exemple d'entree

```
5
EXXXa
*XX*.
XXX.X
XX.AX
XX..S
10
```

On attend, par exemple, la sortie suivante :

Listing 6: Exemple de sortie

```
BAS
TP
DROITE
HAUT
BAS
GAUCHE
BAS
BAS
BAS
DROITE
```

Cette solution est réalisable car elle prend un temps de 9, ce qui est bien inférieur ou égal à 10.

2.3.3 Tester votre solution

Comme expliqué en début de sujet, un script nommé `exec3.sh` vous est proposé pour tester votre code.

Ce script, comme les deux autres, compile votre code et le test sur les instances données en argument. Il existe 18 instances numérotées 01, 02, ..., 17, 18.

- Si vous ne donnez pas d'argument, le script lance votre code sur tous les tests et vous donne un résumé sans rien afficher d'autre.
- Si vous donnez un ou plusieurs arguments, alors le script lance votre code sur chacunes des instances données en argument et vous propose d'animer votre solution. Ceci, vous permettant de comprendre pourquoi votre solution est fausse si c'est le cas. Attention, si votre code met trop de temps

à calculer ou si une erreur de segmentation apparaît alors aucun affichage ne vous est proposé.

- Vous pouvez également préciser l'option `-q`. Dans ce cas, le script n'affiche que le résumé des tests.