Algorithme du plus court chemin

Projet Langage C

Étant donné un graphe orienté pondéré (avec un poids sur les arcs), il existe un algorithme optimal qui trouve le chemin qui minimise le poids total entre un point de départ et un point d'arrivée. Cet algorithme [1] a été proposé par Edsger Dijkstra en 1956 et est toujours utilisé de nos jours dans les tâches de type planification de chemin (à quelques heuristiques près), puisqu'il n'en existe pas de mieux !

1 Définitions et notations

Un graphe orienté $G = \langle N, E \rangle$ se compose d'un ensemble de noeuds (N) et d'arêtes (E). Une arrête $e \in E$ consiste en un noeud de départ $src(e) \in N$ et un noeud d'arrivée $tgt(e) \in N$. Le graphe est dit *orienté* car s'il existe une arrête de A vers B, il n'est pas nécessaire qu'il existe une arrête dans l'autre sens.

Chaque noeud est séparé par une *distance* (orientée), notée $d(n_1, n_2)$ (avec $n_1, n_2 \in N$). Si les deux noeuds ne sont pas reliés par un arc, alors la distance qui les sépare est infinie par convention ($d(n_1, n_2) = \infty$ si $\nexists e \in E, src(e) = n_1, tgt(e) = n_2$).

Soit un noeud $n \in N$, l'ensemble des voisins de n est noté $neigh(n)$, et correspond à l'ensemble des noeuds tel qu'il existe une arrête e de source $src(e) = n$ ($neigh(n) = \{n_v \mid n_v \in N, \exists e \in E, src(e) = n \wedge tgt(e) = n_v\}$).

2 Algorithme de Dijkstra

Soit un graphe $G = \langle N, E \rangle$ et deux noeuds de départ et d'arrivée $n_d, n_a \in N$. L'algorithme de Dijkstra consiste à calculer la distance minimale entre le noeud de départ et chaque noeud du graphe, en enregistrant au passage de quel noeud on vient pour minimiser cette distance (qu'on appellera le *noeud précédent* du noeud en question). Pour cela, on utilise deux collections (des listes chaînées, dans notre cas) : la collection des noeuds *à visiter* ($AVisiter$) et la collection des noeuds *déjà visités* ($Visités$). Chaque noeud n de ces collections stock :

1. la distance calculée pour arriver à ce noeud depuis le noeud de départ ($dist_{AVisiter}(n)$, $dist_{Visités}(n)$)
2. le noeud précédent associé ($prec_{AVisiter}(n)$, $prec_{Visités}(n)$).

Si un noeud n'est pas dans une collection, la distance associée à ce noeud est infinie par convention ($n \notin AVisiter \Rightarrow dist_{AVisiter}(n) = \infty$). Si le noeud n'a pas de précédent (parce qu'il n'est pas dans la collection ou parce que c'est le noeud de départ), on notera $prec_{AVisiter}(n) = *$.

L'idée est que, pour tout noeud de la collection *Visités*, la distance associée à ce noeud est bien la distance minimale, et le noeud précédent associé est celui dont on provient en empruntant le plus court chemin. La distance associée au noeud de destination est alors la plus courte distance, et en remontant le chemin à partir du noeud de destination, on retrouve le plus court chemin entre la source et la destination.

L'algorithme se découpe donc en deux partie : calculer les distances minimales pour chaque noeud (algorithme "D") et calculer le chemin entre n_d et n_a (algorithme "C").

Voici les étapes pour la partie "D" :

(D-1) Ajouter à $AVisiter$ le noeud de départ n_d, avec comme distance 0 et comme précédent $*$

(D-2) Tant qu'il existe un noeud dans $AVisiter$, faire :

- **(D-2.1)** Chercher un noeud $n_c \in AVisiter$ dont la distance associée $dist(n)$ est minimale ; on appelle n_c le *noeud courant*
- **(D-2.2)** Ajouter n_c dans *Visités*, avec $dist_{Visités}(n_c) = dist_{AVisiter}(n_c)$ et $prec_{Visités}(n_c) = prec_{AVisiter}(n_c)$
- **(D-2.3)** Supprimer n_c de $AVisiter$
- **(D-2.4)** Pour chaque voisin de n_c, $n_v \in neigh(n_c)$, faire :
  - **(D-2.4.1)** Calculer la distance totale du noeud de départ n_d à n_v en passant par n_c : $\delta' = dist_{Visités}(n_c) + d(n_c, n_v)$

1. https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

(D-2.4.2) Soit $\delta = dist_{AVisiter}(n_v)$, la distance actuelle de n_d à n_v (potentiellement infinie si n_v n'est pas dans $AVisiter$)

(D-2.4.3) Si $\delta' < \delta$, alors n_c est un meilleur précédent pour n_v : changer les valeurs associée à n_v dans $AVisiter$ de façon à enregistrer le noeud courant comme précédent de n_v et δ' comme nouvelle distance pour n_v : $dist_{AVisiter}(n_v) = \delta'$, $prec_{AVisiter}(n_v) = n_c$
(on notera que si $n_v \notin AVisiter$, cette dernière étape ajoute n_v à $AVisiter$)

Une fois que la boucle est terminée, *Visités* contient tous les noeuds atteignables depuis n_d, et pour tout $n \in Visits$, $dist_{Visités}(n)$ est optimal. En particulier, $dist_{Visités}(n_a)$ est la distance minimale entre n_d et n_a.

On peut maintenant construire le chemin qui va de n_d à n_a, en partant de n_a et en ajoutant récursivement le précédent du dernier noeud ajouté. Pour cela, on utilise une collection $Chemin$, du même type que $AVisiter$ et *Visités* (donc où chaque noeud est associé à un $dist_{Chemin}$ et un $prec_{Chemin}$).

Soit le chemin courant $Chemin$ et le noeud que l'on visite n. On construit le chemin de n_d à n avec l'algorithme "C" :

(C-1) Soit $n_p = prec_{AVisiter}(n)$

(C-2) Si $n_p = *$, on s'arrête ici car normalement $n = n_d$

(C-3) Sinon :

(C-3.1) Construire le chemin de n_d à n_p (appel récursif)

(C-3.2) Ajouter au chemin le noeud n_p (avec $dist_{Chemin}(n_p) = dist_{Visités}(n_p)$ et $prec_{Chemin}(n_p) = prec_{Visités}(n_p)$)

Pour construire tout le chemin, on appelle l'algorithme C avec $n = n_a$.

3 Travail demandé

Le but du projet est de développer en C l'algorithme de Dijkstra. Vous est fourni un module `graphe` qui représente un graphe orienté où les noeuds ont une position cartésienne, et donc où on peut récupérer la distance entre deux noeuds du graphe (qui vaut la distance euclidienne entre ces noeuds s'il existe un arc qui les relie, ou `INFINITY`, la constance C représentant un réel infini sinon).

Pour pouvoir construire l'algorithme, nous aurons besoin d'une collection pour $AVisiter$, *Visités* et $Chemin$; il s'agira de *listes chaînées*, que vous aurez à développer également.

Enfin, le reste des modules fournis permettent de lancer l'algorithme et d'afficher le résultat obtenu dans une fenêtre.

Le projet se base sur un `Makefile`. On peut faire `make help` pour afficher les cibles disponibles.

3.1 Le module `graphe`

Le module `graphe` vous est donné. Il est extensivement documenté : vous pouvez faire `make doc` pour créer la documenation du projet, qui se trouve alors dans le dossier `doc`.

Ce module définit un type pour les graphes : `struct graphe_t`, dont l'implémentation est cachée. On fait référence aux noeuds en utilisant un identifiant, de type `noeud_id_t`. Cet identifiant est ce qui permet de requêter le graphe : récupérer la position d'un noeud, la distance entre deux noeuds, etc. Dans la suite, c'est le type que nous utiliserons dès que nous devrons faire référence à un noeud.

Par convention, le module utilise l'identifiant spécial `NO_ID` pour signifier l'absence de noeud/un noeud qui n'existe pas. Par exemple dans la suite, pour indiquer qu'un noeud n'a pas de prédécesseur, on utilisera `NO_ID` (qui joue donc le rôle de $*$ dans l'énoncé).

Le module définit plusieurs fonctions utiles pour l'algorithme de Dijkstra, en particulier :

— `noeud_distance(graphe, na, nb)` : calcule la distance entre les noeuds `na` et `nb` ; s'ils sont voisins, il s'agit de la distance euclidienne entre les deux noeuds, sinon la fonction renvoie la valeur spéciale `INFINITY`, qui représente l'infini (exactement comme d dans l'énoncé).

— `nombre_voisins(graphe, n)` : recupère le nombre de voisins du noeud n dans le graphe (le cardinal de $neigh(n)$).

— `noeuds_voisins(graphe, n, voisins)` : remplit le tableau `voisins` (tableau de `noeud_id_t`) avec les voisins du noeud `n` (correspond donc à $neigh(n)$) ; le tableau `voisins` passé en paramètre doit avoir été alloué auparavant, avec une taille suffisante pour contenir tous les voisins :

```
// Code (partiel) pour récupérer les voisins
size_t nvoisins = nombre_voisins(graphe, n);
noeud\_id\_t* voisins = /* allocation d'un tableau de nvoisins noeuds */;
noeuds_voisins(graphe, n, voisins);
//...
// ne pas oublier de libérer la mémoire quand on en a plus besoin...
```

3.2 Le module `liste_noeud`

Dans un premier temps, nous allons développer le module `liste_noeud`, qui définit un type pour une liste de noeuds, ainsi que toutes les fonctions dont nous aurons besoin pour réaliser l'algorithme et permettre à l'application d'afficher le résultat.

Attention : le nom des fonctions et l'ordre des arguments est à respecter *impérativement*.
Avant de compiler tout le projet, utilisez le script `test_struct` pour vérifier que votre module `liste_noeud` est conforme. Vous pouvez aussi lancer le script avec `make test_struct`. N'oubliez pas de rendre le script exécutable s'il ne l'est pas déjà, avec `chmod u+x test_struct`.

Le module `liste_noeud` définit une liste chaînée, où chaque cellule contient un noeud, une distance et un noeud précédent (n, $dist(n)$, $prec(n)$).

Pour commencer, complétez le fichier `liste_noeud.h` de manière à spécifier le module `liste_noeud`, en veillant bien à ce que le fichier respecte les contraintes et bonnes pratiques vues en cours. Des contrats exhaustifs vous sont fournis, à respecter absolument. Ce fichier contient les déclarations suivantes :

— un type enregistrement `liste_noeud_t` **abstrait** qui représente une liste de noeuds, avec un `typedef` pour faciliter l'écriture du reste du module
— la fonction `creer_liste` pour créer un `liste_noeud_t` et une fonction `detruire_liste` pour en disposer et libérer la mémoire associée
— la fonction `est_vide_liste` qui test si une liste est vide, `contient_noeud_liste` qui vérifie si un noeud appartient à la liste, et `contient_arrete_liste` qui vérifie si une arête appartient à la liste (ces deux dernières déclarations sont données car elles sont utilisées dans du code fourni)
— les fonctions `distance_noeud_liste` et `precedent_noeud_liste`, qui récupèrent (respectivement) la distance et le noeud précédent associé à un noeud donné, ou une valeur par défaut *documentée dans le contrat* si le noeud n'existe pas dans la liste
— la fonction `min_noeud_liste`, qui récupère le noeud de la liste dont la distance associée est la plus petite (ou renvoie `NO_ID` si la liste est vide)
— les fonctions `inserer_noeud_liste` pour ajouter un noeud dans la liste, `changer_noeud_liste` pour modifier les données associées à un noeud dans la liste (ou l'ajouter s'il n'existe pas) et `supprimer_noeud_liste` pour supprimer un noeud de la liste

Vous veillerez à marquer les paramètres comme `const` lorsque c'est approprié.

Une fois que vous avez complété le fichier `liste_noeud.h`, on peut passer à l'implantation, dans le fichier `liste_noeud.c`.

Vous devez fournir une implantation du type abstrait `liste_noeud_t` et de chaque fonction déclarée dans le header. Le type `liste_noeud_t` représente une *liste chaînée*.

Pour rappel, une liste chaînée est une suite de *cellules*. Dans notre cas, chaque cellule contient un noeud, une distance, un noeud précédent et le pointeur sur la cellule suivante. La liste est alors un simple enregistrement, qui contient un pointeur sur la première cellule, et éventuellement un pointeur sur la dernière cellule pour plus d'efficacité (optionnel).

Nous vous suggérons de d'abord définir un type *interne* `_cellule` qui représente une cellule, et de réaliser l'implantation de `liste_noeud_t` à l'aide de ce type.

C'est à vous de choisir la variante exacte de liste chaînée que vous implanterez (avec/sans sentinelle, ordonnée, pointeur sur début et fin/pointeur sur début uniquement, etc.). Les tests unitaires fournis sont écrits

indépendamment de tout type d'implantation (normalement).

Encore une fois, vous veillerez à respecter scrupuleusement les contrats fournis ; les tests unitaires explorent les pré-conditions au maximum !

3.3 Le module `dijkstra`

Le module `dijkstra` définit la fonction `dijkstra`, qui réalise l'algorithme du plus court chemin de Dijkstra. Le header `dijkstra.h` déclare la fonction, dont le prototype est rappelé ici :

```
float dijkstra(
        const struct graphe_t* graphe,
        noeud_id_t source, noeud_id_t destination,
        liste_noeud_t** chemin);
```

Cette fonction prend 4 paramètres :
- `graphe` : le graphe dans lequel l'algorithme se déroule (entrée)
- `source` et `destination` : le noeud de départ et d'arrivée (respectivement) du chemin à calculer
- `chemin` : un pointeur vers une liste de noeuds dans laquelle l'algorithme va enregistrer le chemin calculé (sortie) ; c'est à la fonction de créer la liste (avec `creer_liste`) en plus de la remplir

L'implantation de la fonction `dijkstra` se fait en deux parties : le calcul des distances dans le graphe (étapes "D" dans l'énoncé) et la récupération du plus court chemin (étapes "C" dans l'énoncé).

Dans un premier temps, implantez l'algorithme de Dijkstra "D" tel que présenté dans l'énoncé, en utilisant les fonctions du module `liste_noeud` développé et les fonctions du module `graphe` présenté dans la section "Le module `graphe`" (voir plus haut).

Dans un second temps, implantez la fonction récursive `construire_chemin_vers` spécifiée dans `dijkstra.c`, qui construit un chemin en suivant l'algorithme "C" de l'énoncé.

La fonction `dijkstra` doit, après avoir complété la partie "D" de l'algorithme, et si le paramètre `chemin` ne vaut pas NULL, créer un `liste_noeud_t` pour le chemin, et utiliser `construire_chemin_vers` pour enregistrer le chemin.

4 Tests et applications

Un Makefile vous est fourni, qui vous permet de compiler des parties du projet et de lancer des tests :
- Pour vérifier la conformance de vos fichiers : `make test_struct`
- Pour compiler les tests unitaires de `liste_noeud` : `make test_liste_noeud`
- Pour lancer les tests unitaires de `liste_noeud` : `make runtest_liste_noeud`
- Pour compiler les tests unitaires de `dijkstra` : `make test_dijkstra`
- Pour lancer les tests unitaires de `dijkstra` : `make runtest_dijkstra`
- Pour lancer tous les tests unitaires avec Valgrind (voir plus bas) : `make runtests_valgrind`
- Pour compiler l'application principale : `make dijkstra` ou `make dijkstra_debug` pour avoir accès à des messages de debug
- Pour créer l'archive à rendre : `make rendu`
  Vous devez impérativement utiliser cette commande pour créer l'archive, sinon la correction ne se passera pas bien

4.1 L'application `dijkstra`

L'application `dijkstra` utilise votre code pour calculer et afficher un plus court chemin. Elle se lance avec trois arguments : un fichier `.graphe` qui contient une description de graphe, le nom du noeud de départ et le nom du noeud d'arrivée. Nous vous fournissons deux fichiers `.graphe` pour tester : `test.graphe`, un graphe assez simple pour vérifier si ça foncitonne, et `occitanie.graphe`, un graphe qui représente des villes d'Occitanie (avec des positions tirées des coordonnées GPS respectives de ces villes).

Par exemple, pour montrer le chemin le plus court de Revel à Tarbes :

```
> ./dijkstra occitanie.graphe Revel Tarbes
```

N'hésitez pas à expérimenter avec vos propre `.graphe` en vous inspirant de la syntaxe des fichiers donnés (la syntaxe est aussi expliquée plus en profondeur dans la documentation du module `graphe_parse`).

4.2 Utilisation de Valgrind pour les erreurs de mémoire

Le projet fait beaucoup appel à l'allocation dynamique (pour les listes chaînées et pour l'algorithme de Dijkstra lui-même), ce qui est une source importante d'erreurs. Nous attendons de vous que vous soyez *très rigoureux* sur la gestion de mémoire (allocation, libération). En particulier, votre code ne doit pas présenter d'erreurs de mémoire (fatale ou non) ni de fuite de mémoire.

Pour vérifier si votre programme présente des erreurs, nous utiliserons l'outil Valgrind [2]. La vérification est réalisée sur les tests unitaires comme suit :

```
> valgrind -s --leak-check=full ./test_liste_noeud
> valgrind -s --leak-check=full ./test_dijkstra
```

Ou plus simplement avec le Makefile :

```
> make runtests_valgrind
```

Nous vous recommandons de lancer systématiquement les tests unitaires avec Valgrind, car il peut révéler la source d'erreurs de segmentation que votre code pourrait contenir !

4.3 Pour aller plus loin...

Si vous êtes intéressé.e par quelques fonctionnalités avancées du langage C, vous pouvez consulter le classeur `LangageC_2-4`, qui propose une petite amélioration au projet. Ceci est bien entendu *optionnel* ; commencez par avoir un projet fonctionnel avant de vous aventurer sur des territoires inconnus !

5 Modalités de rendu et d'évaluation

Le rendu se fait **sur Moodle**, sur la page de la matière à l'endroit indiqué. Il ne doit y avoir **qu'un seul rendu par groupe**.

Vous devrez déposer sur Moodle une **archive .tar.gz** que l'on obtient en utilisant le script fourni, `creer_rendu`. Pour cela, vous pouvez simplement faire : `./creer_rendu` ou `make rendu`. La commande vous pose des questions, puis crée un fichier `rendu.tar.gz`, que vous n'avez plus qu'à déposer sur Moodle.

Pour votre information, l'archive contiendra les fichiers `liste_noeud.h`, `liste_noeud.c`, `dijkstra.h` et `dijkstra.c`, ainsi qu'un fichier d'identification pour faire le lien entre l'archive et le groupe. Il ne vous est pas permis de rendre des fichiers additionnels ou d'autres fichiers que vous auriez modifié.

Lors de la correction, les fichiers sont placés dans un environnement contrôlé (le même que celui qui vous est fourni, en fait) et testés dans ces conditions avec les tests unitaires qui vous ont été fournis. L'outil Valgrind sera utilisé pour détecter les erreurs et fuites de mémoire.

Nous attendons de vous que :

1. **vous respectiez scrupuleusement les consignes énoncées dans cette partie (utilisation du script, complétion de A-COMPLETER.txt)**
2. **les fichiers rendus compilent *sans erreur* dans l'environnement (celui des machines de l'ENSEEIHT)**

En cas de non respect de l'une de ces deux règles, **la note sera automatiquement mise à 0**.

Par ailleurs, notez que **l'utilisation de modèles de langage (ChatGPT, Copilot, etc.) est strictement interdite, de même que la recopie depuis un autre groupe**. Tout triche entraînera des sanctions.

2. `https://valgrind.org/`

Les rendus sont compilés, testés et examinés par votre intervenant de TP. Ils sont évalués selon 3 critères principaux :

1. Fonctionnalité du code
2. Respect des consignes
3. Qualité du code

Pour avoir le maximum de points, il faut notamment que la compilation n'entraîne aucun warning (les options `-Wall -Wextra -pedantic` sont utilisées), et que *tous les tests unitaires* passent. Les contrats doivent être complets et exhaustifs (et respecter la spécification présentée dans le sujet du projet), et le code doit respecter les bonnes pratiques de la programmation que vous devez maîtriser (indentation correcte, nom de variable qui a du sens, utilisation des bonnes structures, etc.).