Algorithme de Dijkstra

Jean-Baptiste Rouquier

1 graphes

Definition. Un ***graphe orienté*** fini est un couple (V, E) où V est un ensenble fini et E une partie de $V \times V \setminus \{(v, v) \mid v \in V\}$.

V est l'ensemble des ***sommets*** (en anglais *vertices*), E (*edges*) celui des arêtes.
En pratique on ne précise pas fini.
$|V|$ est noté n, $|E|$ est noté m. On identifie ici $[\![0, n-1]\!]$ et V :

```
type sommet = int
```

Definition. Un ***graphe non-orienté*** est un couple (V, E) où V est un ensemble (fini par défaut) et E un sous-ensemble des parties à deux éléments de V.

Question 1.1. Exprimer les graphes orientés et non-orientés en termes de relations.

La suite du TP n'utilise que des graphes orientés.

Definition. Un ***chemin*** est une suite de sommets $(v_0, \ldots, v_k)$ telle que $\forall\ 0 \leqslant i < k \quad (v_i, v_{i+1} \in E)$ et $\forall\ 0 \leqslant i < j < k \quad (v_i, v_{i+1}) \neq (v_j, v_{j+1})$. C'est un ***cycle*** si $v_0 = v_k$. k est la ***longueur*** du chemin, c'est aussi son nombre d'arêtes.

Remarque. On parle de ***chaîne*** dans les graphes non orientés.

Definition. Le ***voisinage*** du sommet x est $\{y \mid (x, y) \in E\}$.

Il y a deux structures de données usuelles pour représenter un graphe :
- Soit on mémorise, pour chaque couple (i, j), s'il y a un arc de i à j. On utilise donc une matrice dont la case de coordonnées (i, j) contient **true** si et seulement si il existe un arc de i à j.
- Soit on stocke les voisins de i pour tout i, on utilise alors un tableau dont la case i contient la liste des j tels que l'arête (i, j) appartienne au graphe.

```
type graph_matrix = bool array array
type graph_list = sommet list array
```

L'une ou l'autre sera dans la suite notée `graph`, vous devrez choisir celle qui vous semble la plus adaptée.

Question 1.2. Dessiner quelques graphes (différents de ceux des voisins) et les encoder dans chacune des structures de données.

Ces graphes seront utilisés pour tester les questions suivantes, faites en au moins deux distincts d'au moins 6 sommets et 6 arêtes.

2 L'algorithme de Dijkstra

Son but est calculer le plus court chemin d'un sommet s (*source*) donné à un autre sommet t (*target*) donné.

Il consiste à répartir les sommets en trois groupes :
- `examined` est l'ensemble des sommets v pour lesquels on connaît le plus court chemin de s à v. Initialement c'est $\varnothing$.

– `current` est l'ensemble des sommets voisins d'un sommet examiné, mais pas eux-mêmes examinés.
– `unknown` contient tous les autres sommets.

On pose donc

```
type status = Examined | Current | Unknown
```

Question 2.1. Choisir un type (impératif)[1] `groups` pour représenter les trois groupes dans une seule variable. Il doit permettre d'obtenir le groupe d'un sommet et de changer un sommet de groupe.

Ecrire les fonctions

```
initialize : graph -> groups
get_group : groups -> sommet -> status
change_group : groups -> sommet -> status -> unit
```

Question 2.2 (bonus). Quels seraient les types de ces trois fonctions si on choisissait plutôt une structure de données fonctionnelle ?

On stocke pour chaque élément v de `current` la longueur $c(v)$ du plus court chemin de s à v passant uniquement par des sommets de `examined` (sauf bien sûr le dernier, v, qui n'est pas dans `examined` mais dans `current`). Les valeurs de c seront stockées dans un tableau de type `int option array`. La case i contient `Some c(`i`)` si $i \in$ `current` $\cup$ `examined`, `None` sinon (c'est-à-dire si on ne connaît pas $c(i)$).

À chaque étape, on choisit v dans `current` minimisant $c(v)$.

Question 2.3. Montrer que pour ce sommet v, $c(v)$ est la longueur du plus court chemin de s à v (faire un dessin).

Question 2.4. On pose `type longueur = int`.

Choisir une structure de données pour représenter l'ensemble des sommets de `current` (il y a certes redondance avec `groups`). Elle doit permettre de trouver l'élément v minimisant $c(v)$ et d'ajouter des éléments à la structure. Commencer par une structure simple, optimiser s'il reste du temps. Écrire les valeurs (fonctionnelles) :

```
empty : current
take_minimum : current -> sommet * current
add : sommet -> longueur -> current -> current
```

On connaît la longueur du plus court chemin de s à v (cf. question 2.3), on déclare donc ce sommet v comme examiné (par un appel à `change_group`). On met ensuite à jour `current` et c.

Question 2.5. Expliciter les opérations nécessaires pour mettre à jour current et c.

Question 2.6. Déterminer une condition d'arrêt.

Question 2.7. Implémenter l'algorithme.

```
dijkstra : graph -> sommet -> sommet -> longueur
```

Question 2.8. Vérifiez votre code sur les graphes de la question 1.2.

Question 2.9. Montrer que l'algorithme ainsi décrit est correct et termine.

Question 2.10. Que se passe-t-il si le graphe contient des cycles ?

Question 2.11 (bonus). Récrire les structures de données et l'algorithme précédent pour retourner le chemin (sous forme de liste de sommets) le plus court, et non seulement sa longueur.

Question 2.12 (bonus). On considère maintenant une fonction de poids $w : E \to \mathbb{N}$. Le poids d'un chemin est la somme des poids de ses arêtes. Récrire l'algorithme précédent pour calculer un chemin de poids minimal de s à t.

Question 2.13 (bonus). Que se passe-t-il s'il y a des poids négatifs ?

[1] Une structure de données est ***impérative*** si elle est modifiée en place (comme les tableaux ou les références) : appliquer une fonction à une telle variable change sa valeur et renvoie souvent `unit`.

Les structures de données fonctionnelles sont au contraire non modifiables (comme les listes ou les types sommes). Appliquer une fonction à une telle variable de type `foo` ne peux pas la modifier (utile si elle est utilisée ailleurs) et renvoit souvent une nouvelle variable de type `foo`.