Graphes pondérés et files de priorité : algorithme de Dijkstra

14 décembre 2023

Considérons un graphe $G = (V, E)$ connexe.
On peut pondérer les arêtes du graphe avec une fonction $p : E \to \mathbb{R}$, dite de pondération.
On considère maintenant un graphe non orienté G connexe et pondéré à poids positifs.
Dans tout ce TP, on utilisera le type suivant :

```
type graphe = (int*int) list array
```

1 Implémentation de Dijkstra à l'aide d'une file de priorité

Voici une version de Dijkstra :

1. On initialise une file de priorité vide.
2. On initialise un tableau `dist` des distances indexé par les sommets, on met la distance de `s` (l'origine) à 0 et pour tous les autres sommets à $+\infty$.
3. On insère tous les sommets dans la file de priorité avec la valeur associée dans le tableau des distances comme priorité.
4. Tant que la file n'est pas vide :
   (a) on extrait `x` le sommet de poids (valeur dans le tableau distance) minimal.
   (b) pour tous les voisins `y` de `x` : si `dist.(y)>dist.(x)+poids(x,y)` alors on met à jour `dist.(y)` ainsi que l'organisation de la file de priorité en conséquence.
5. on renvoie le tableau `dist`.

L'avantage de cette version est que la preuve de sa correction est plus confortable et que sa complexité en espace est optimisée : la taille de la file de priorité est majorée par le nombre de sommets alors que dans la version simplifée de la partie suivante elle l'est par deux fois le nombre d'arêtes.

L'inconvénient est qu'elle est un peu plus technique à implémenter. Puisque les sommets n'apparaissent qu'une seule fois dans la file de priorité, il est nécessaire de mettre à jour leur valeur de priorité et de réorganiser la file. Cette mise à jour implique une réorganisation de la file de priorité ce qui n'est pas trop difficile à condition de savoir où se trouve le noeud dans le tas. Pour cela, on devra donc maintenir un tableau `tab_pos` qui retiendra dans quelle case du tableau représentant le tas (ie la file de priorité) se trouve chacun des sommets du graphe. Il va donc falloir adapter les fonctions de gestion de la file de priorité car chaque opération de file impliquera une modification éventuelle du tableau `tab_pos`.

1.1 Files de priorité :

On rappelle qu'une file de priorité est une structure de donnée qui permettra facilement de récupérer un élément de priorité (ici) minimale, de supprimer cet élément et d'insérer un nouvel élément.

On considère donc le type suivant :

```
type heap = {mutable taille : int; values : (int*int) array}
```

▷ **Question 1.** Ecrire une fonction `swap : int*int array -> int->int-> int array->unit` telle que `swap tableau pos1 pos2 tab_pos` échange le contenu des cases d'indices `pos1` et `pos2` de `tableau` et met à jour le tableau `tab_pos` pour que dans sa case d'indice `k` se trouve la position de l'élément de type `(k,_)` dans `tableau`. On suppose que le tableau `tab_pos` contenait déjà les positions des éléments de `tableau` et on met

à jour les positions de ceux qui ont été échangés. On suppose aussi qu'il n'y a qu'un seul élément de la forme `(k,_)`. ◁

▷ **Question 2.** Ecrire une fonction `insere : int*int->heap->int array->unit` telle que `insere x tas tab_pos` insère le couple `x` dans `tas` en mettant à jour le tableau des positions pour qu'il reste cohérent avec les modifications effectuées dans le tas. ◁

▷ **Question 3.** Ecrire une fonction `delete_min : heap->int array->int` telle que `delete_min tas tab_pos` supprime l'élément de poids (deuxième valeur du couple) minimal, mette à jour le tableau `tab_pos` et renvoie la valeur (première valeur du couple) de poids minimal. ◁

▷ **Question 4.** Ecrire une fonction `modif_poids : heap -> int->int ->int array-> unit` telle que `modif_poids tas s new_poids tab_pos` modifie **à la baisse** le poids associé au sommet `s` et mette à jour le tableau `tab_pos` en fonction des modifications apportées à l'organisation du tas. ◁

1.2 L'algorithme de Dijkstra

▷ **Question 5.** Ecrire une fonction `dijkstra : graphe->int->int array` telle que `dijkstra g s` renvoie un tableau tel que dans la case d'indice `i` se trouve la distance minimale d'un chemin de `s` à `i` dans `g`. On utilisera les fonctions de la partie précédente afin de gérer la file de priorité contenant les sommets. ◁

2 Dijkstra : une deuxième version

La version de Dijkstra la plus simple à écrire consiste à adapter le parcours en largeur dont on rappelle ici le principe :

1. on insère le sommet d'origine `s` dans une file `F`
2. Tant que `F` est non vide :
   (a) on extrait `x` le premier élément de la file et s'il n'est pas marqué alors on le marque et on enfile tous ses voisins dans la file.

On remarque que les éléments sont potentiellement insérés plusieurs fois dans la file mais seule la première occurence sera considérée grâce au marquage.

▷ **Question 6.** Ecrire une fonction `parcours_largeur : graphe -> int -> unit` telle que `parcours_largeur g s` parcourt `g` en largeur à partir du sommet `s` et affiche (utiliser `print_int`) ses sommets. On pourra utiliser le module `Queue` de Ocaml. ◁

Pour adapter ce parcours, il suffit de remplacer la file par une file de priorité en utilisant comme valeur de priorité pour un sommet `x` la distance actuellement estimée par l'algorithme entre `x` et l'origine du parcours `s`. On peut considérer une file de priorité qui contient les couples (sommet,distance estimée). Lorsqu'on insère un voisin `y` de `x` dans la file de priorité alors on le fait avec un poids qui sera $p(x,y)+d(x)$ et s'il y a plusieurs occurrences de y dans la file c'est celle qui correspond à la plus petite valeur de distance qui sortira en premier et qui sera considérée.

▷ **Question 7.** En utilisant le fichier `file_prio.ml`, écrire une fonction `dijkstra : graphe -> int -> int array` telle que `dijkstra g s` renvoie un tableau indexé par les sommets de `g` et contenant la distance à `s` pour chacun d'entre eux. ◁