

# Graphes : modélisation et parcours

OPTION INFORMATIQUE - TP n° 3.2 - Olivier Reynet

## À la fin de ce chapitre, je sais :

- ✎ modéliser un graphe par liste d'adjacence
- ✎ modéliser un graphe par matrice d'adjacence
- ✎ passer d'une modélisation à une autre
- ✎ parcourir un graphe en largeur et en profondeur
- ✎ implémenter l'algorithme de Dijkstra

## A Modélisation d'un graphe

Dans ce qui suit on peut considérer le graphe :

```
1  let g = [ | [1;2] ; [0;3;4] ; [0;5;6] ; [1] ; [1] ; [2] ; [2] | ] ;;
```

- A1. Sous quelle forme le graphe g est-il donné?
- A2. Dessiner le graphe g. Comment peut-on qualifier ce graphe?
- A3. On dispose d'un graphe sous la forme d'une liste d'adjacence. Écrire une fonction `list_to_matrix` qui transforme cette représentation en une matrice d'adjacence.
- A4. On dispose d'un graphe sous la forme d'une matrice d'adjacence. Écrire une fonction `matrix_to_list` qui transforme cette représentation en une liste d'adjacence.
- A5. On dispose d'un graphe orienté sous la forme d'une liste d'adjacence. Écrire une fonction `desoriented_list` qui transforme ce graphe en un graphe non orienté.
- A6. On dispose d'un graphe orienté sous la forme d'une matrice d'adjacence. Écrire une fonction `desoriented_matrix` qui transforme ce graphe en un graphe non orienté.

## B Parcourir un graphe

Le parcours d'un graphe est une opération fondamentale et utilisée par de nombreux algorithmes, notamment Dijkstra et A\*. On peut facilement mémoriser les différentes stratégies en observant les types d'ensemble qui sont utilisés pour stocker les sommets à parcourir au cours de l'algorithme :

1. Le parcours en **largeur** passe par tous les voisins d'un sommet avant de parcourir les descendants de ces voisins. Les sommets passent dans une **file** de type First In First Out.
2. Le parcours en **profondeur** passe par tous les descendants d'un voisin d'un sommet avant de parcourir tous les autres voisins de ce sommet. Les sommets passent dans une **pile** de type Last In First Out.

3. L'algorithme de **Dijkstra** passe par le voisin le plus proche d'un sommet avant de parcourir les autres voisins de ce sommet. C'est un parcours en largeur qui utilise une **file de priorités** : lorsqu'on insère un nouvel élément dans cette file, celui-ci est placé d'après son niveau de priorité, le plus prioritaire en premier. Dans notre cas, la priorité est la distance. La plus petite distance en tête donc.

**Dans cette section, on suppose qu'on manipule un graphe sous la forme d'une liste d'adjacence.**

- B1. Écrire une fonction récursive de signature `bfs : int list array -> int -> int list` qui parcourt en largeur un graphe et qui renvoie la liste des sommets parcourus. On pourra s'inspirer du code suivant :

```

1  let bfs g v0 =
2    let visited = Array.make (Array.length g) false in
3    let rec explore queue = (* queue -> file en anglais FIFO *)
4      match queue with
5      | ..
6    in explore [v0] ;;

```

Tester l'algorithme sur le graphe suivant :

```

1  let g = [| [1;2] ; [0;3;4] ; [0;5;6] ; [1] ; [1] ; [2] ; [2] |] ;;

```

- B2. Écrire une fonction récursive de signature `dfs : int list array -> int -> int list` qui parcourt en profondeur un graphe et qui renvoie la liste des sommets parcourus.

## C Plus courts chemins : algorithme de Dijkstra

---

**Algorithme 1** Algorithme de Dijkstra, plus courts chemins à partir d'un sommet donné

---

1: <b>Fonction</b> DIJKSTRA( $G = (V, E, w), a$ )	▷ Trouver les plus courts chemins à partir de $a \in V$
2: $\Delta \leftarrow a$	▷ $\Delta$ est l'ensemble des sommets dont on connaît la distance à $a$
3: $\Pi \leftarrow \emptyset$	▷ $\Pi[s]$ est le parent de $s$ dans le plus court chemin de $a$ à $s$
4: $d \leftarrow \emptyset$	▷ l'ensemble des distances au sommet $a$
5: $\forall s \in V, d[s] \leftarrow w(a, s)$	▷ $w(a, s) = +\infty$ si $s$ n'est pas voisin de $a$ , 0 si $s = a$
6: <b>tant que</b> $\bar{\Delta}$ n'est pas vide <b>répéter</b>	▷ $\bar{\Delta}$ : sommets dont la distance n'est pas connue
7:     Choisir $u$ dans $\bar{\Delta}$ tel que $d[u] = \min(d[v], v \in \bar{\Delta})$	
8: $\Delta = \Delta \cup \{u\}$	▷ On prend la plus courte distance à $a$ dans $\bar{\Delta}$
9: <b>pour</b> $x \in \bar{\Delta}$ <b>répéter</b>	▷ Ou bien $x \in \mathcal{N}_G(u) \cap \bar{\Delta}$ , pour tous les voisins de $u$ dans $\bar{\Delta}$
10: <b>si</b> $d[x] > d[u] + w(u, x)$ <b>alors</b>	
11: $d[x] \leftarrow d[u] + w(u, x)$	▷ Mises à jour des distances des voisins
12: $\Pi[x] \leftarrow u$	▷ Pour garder la tracer du chemin le plus court
13: <b>renvoyer</b> $d, \Pi$	

---

- C1. Démontrer la terminaison de l'algorithme de Dijkstra.  
 C2. Démontrer la correction de l'algorithme de Dijkstra.  
 C3. Quelle est la complexité de l'algorithme de Dijkstra?

- C4. Exécuter à la main l'algorithme de Dijkstra sur le graphe orienté suivant en complétant à la fois le tableau des distances et le tableau des parents qui permet de reconstruire le chemin a posteriori. Le tableau parent à la case  $i$  contient le sommet précédent sur le chemin.

```
1  let g = [| [(1,7);(2,1)] ; [(3,4); (5,1)] ; [(1,5);(4,2);(5,7)] ; [] ; [(1,2);(3,5)] ; [(4,3)] |] ;;
```

---

- C5. Compléter le code de la fonction récursive de signature `dijkstra : (int * int)list array -> int array * int array` qui renvoie les plus courtes distances à partir d'un sommet d'un graphe ainsi que les directions à prendre. La fonction `insert_neighbours` renvoie la file de sommets à explorer dans l'ordre d'exploration, la plus petite distance en premier. On pourra utiliser le tri par insertion d'une file pour en faire une file de priorités. La fonction `update_distances_and_parents` met à jour les tableaux de résultats : les distances au sommet de départ et le parent de chaque sommet (pour savoir quel chemin prendre).

```
1  let dijkstra g =
2    let n = Array.length g in
3    let distances = Array.make n max_int and
4      visited = Array.make n false and
5      parents = Array.make n max_int
6    in distances.(0) <- 0; parents.(0) <- 0
7    in let rec insert_neighbours sorted neighbours =
8      ...
9    in let update_distances_and_parents v =
10     ...
11    in let rec explore pq =
12      match pq with
13      | [] -> distances, parents
14      | (v,dv)::t when visited.(v) -> explore t
15      | (v,dv)::t -> visited.(v) <- true;
16                    update_distances_and_parents v;
17                    explore (insert_neighbours t g.(v));
18    in explore [(0,0)];; (*you could choose another one !*)
```

---

- C6. Exécuter l'algorithme de Dijkstra sur le graphe suivant :

```
1  let g = [| [(1,7);(2,1)] ;
2    [(0,7);(2,5);(3,4);(4,2);(5,1)] ;
3    [(0,1);(1,5);(4,2);(5,7)];
4    [(1,4);(4,5)];
5    [(1,2);(2,2);(3,5);(5,3)];
6    [(1,2);(2,7);(4,3)] |] ;;
```

---

Les résultats sont-ils cohérents?

- C7. Est-ce qu'utiliser un tas binaire pour implémenter la file de priorités permettrait d'améliorer la complexité de l'algorithme?