TP 4

Structure de données

Shir-Li Kedem

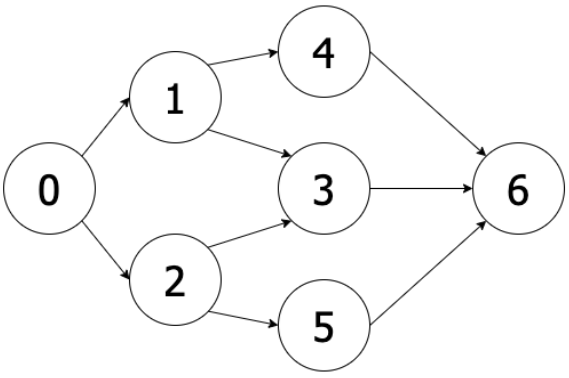
Sciences informatiques

04/05/2023

1.1 Graphe non pondéré

1. Voici un dessin du graphe :

   On a bien 7 nœuds et 9 arêtes.



G = (V,E)
V = {0,1,2,3,4,5,6}
E = {(0,1),(0,2),(1,4),(1,3),(2,3),(2,5),(3,6),(4,6),(5,6)}

2. A gauche : La matrice d'adjacence est carrée et représente les relations entre les nœuds dans un graphe. Chaque case de la matrice contient un 1 s'il y a une arête entre les deux nœuds correspondants, sinon elle contient un 0.
   A droite : La liste d'adjacence est la représentation plus compacte que la matrice d'adjacence. Chaque nœud du graphe a une liste qui contient les nœuds voisins avec lesquels il partage une arête.

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	1	0	0	0	0
1	0	0	0	1	1	0	0
2	0	0	0	1	0	1	0
3	0	0	0	0	0	0	1
4	0	0	0	0	0	0	1
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0

0: [1, 2]
1: [4, 3]
2: [3, 5]
3: [6]
4: [6]
5: [6]
6: []

3. DFS et BFS sont les deux algorithmes utilisés pour parcourir des graphes. Le but est de visiter tous les nœuds du graphe en utilisant les arêtes pour se déplacer d'un nœud à un autre.
   DFS commence par un nœud de départ et explore autant que possible le graphe avant de revenir en arrière.
   ➡️ Il va parcourir le long d'une branche avant de revenir en arrière pour parcourir une autre branche.
   Avantages : Simple à implémenter, nécessite peu de mémoire.
   BFS commence comme DFS, mais explore tous les nœuds du niveau actuel avant de passer au niveau suivant.
   ➡️ Il va parcourir d'abord tous les nœuds à une distance de 1, puis tous les nœuds à une distance de 2, etc.
   Avantages : Trouve le chemin le plus court entre 2 nœuds.

4. Exemple d'application de DFS : On commence à partir du nœud 0
   0 → 1 → 4 → 6 → 3 → 2 → 5
   Exemple d'application de BFS : On commence également à 0
   0 → 1 → 2 → 3 → 4 → 5 → 6

5. Un scénario réel où BFS est préférable à DFS est par exemple la recherche du chemin le plus court dans un réseau de transport. Supposons que nous devons trouver le chemin le plus court entre 2 stations de tram dans le canton de Genève, le graphe qui représente le réseau est très grand avec beaucoup de stations et plusieurs itinéraires possibles pour se rendre à destination. Dans ce cas, BFS parcours toutes les stations accessibles en 1 changement de tram, puis 2 changements, etc, jusqu'à ce que la station de destination soit atteinte, ce qui correspond à l'itinéraire le plus court. En revanche, DFS parcourrait un grand nombre de stations inutiles avant de trouver le chemin le plus court.

1.2 Graphe pondéré

1. Voici la matrice d'adjacence qui prend en compte les distances : Elle représente les connexions entre les nœuds du graphe. Dans ce cas, la matrice d'adjacence est pondérée avec les valeurs de distance pour chaque arête. La valeur 1 indiquant une connexion entre les nœuds et la valeur 0 aucune connexion.

	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	1	0	0	0	0
1	0	0	0	1	1	0	0
2	0	0	0	1	0	1	0
3	0	0	0	0	0	0	1
4	0	0	0	0	0	0	1
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0

2. Cette matrice d'adjacence et ce graphe pourraient représenter un réseau routier reliant plusieurs villes dans une région. Chaque nœud représente une ville, et chaque arête une route entre deux villes. Les pondérations de distance entre les villes peuvent représenter la distance réelle en kilomètres ou en temps de trajet. Cette représentation du réseau routier peut être utile par exemple pour planifier des itinéraires, optimiser des livraisons de marchandises ou bien analyser le trafic routier dans la région.

3. Le principe de fonctionnement d'un algorithme du plus court chemin est de trouver le chemin le plus court entre 2 nœuds dans un graphe pondéré. L'algorithme détermine le chemin le plus optimal en explorant tous les chemins possibles et en comparant leurs distances.
   L'utilité de cet algorithme est qu'il peut être utilisé pour résoudre de nombreux problèmes pratiques comme énoncés dans le point précédent.

4. Pour illustrer l'application d'un tel algorithme sur le graphe donné, supposons que le but est de trouver le chemin le plus court entre le nœud 0 et le nœud 6.

L'algorithme de Dijkstra est pertinent ici :

D'abord l'algorithme initialise tous les nœuds avec une distance infinie sauf le nœud de départ qui a une distance de 0.

Ensuite, l'algorithme parcourt tous les nœuds adjacents et met à jour leurs distances en fonction des distances pondérées entre les nœuds.

Enfin, l'algorithme continue de parcourir les nœuds jusqu'à ce qu'il atteigne le nœud de destination, ou qu'il n'y ait plus de nœuds à parcourir.

➡️ Dans ce cas, l'algorithme de Dijkstra donnerait le chemin le plus court de 0 à 6 qui est : 0 → 1 → 4 → 6. Cela veut dire que le chemin optimal pour aller de la ville "0" à la ville "6" dans notre exemple de réseau routier serait de passer par la ville "1", puis la ville "4" avant d'arriver à la ville "6".

1.1 Modélisation et Implémentation

1.

TGraph
-> Represents oriented graph from adjacency list
Create :
create_graph()
Read :
a. print_list() Prints adjacency list from graph b. add_edge() Add connection between 2 vertices d. populate_graph_from_matrix() Creates adjacency list from matrix

Nous avons ici 3 TAD différents.

1) TGraph est un type qui représente un graphe.
2) TQueue qui représente une queue dans un algorithme qui parcours le graphe.
3) TNode qui représente un nœud dans un graphe.

TQueue
-> Represents a queue in the algorithm exploring the graph
Create :
create_queue()
Modify :
a. add_to_queue() Add element to queue b. remove_from_queue() Removes the last element from queue
Read :
a. is_empty() Verifies if the queue is empty b. print_queue() Prints queue

TNode
-> Represents a node in a graph
Create :
create_node()