Notes de cours
Semaine 20

Cours Turing

Table des matières

1 Rappel - Arbre couvrant

Un arbre couvrant d'un graphe est un sous-graphe qui est un arbre et qui contient tous les sommets du graphe initial.

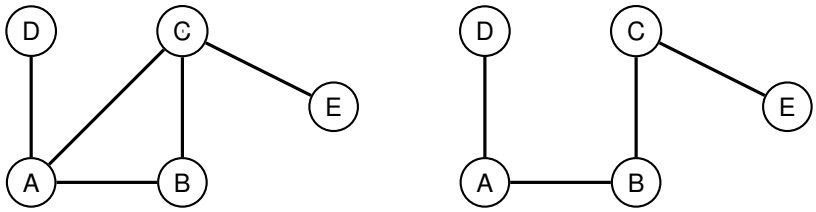


Figure 1: A gauche, un graphe et à droite un arbre couvrant de ce graphe

2 Graphe pondéré

Un graphe pondéré est un graphe où des valeurs numériques sont assignées aux arêtes.

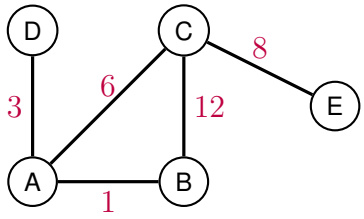


Figure 2: Un graphe pondéré

2.1 Longueur d'un chemin

Dans un graphe non pondéré, la longueur d'un chemin est le nombre d'arêtes qui le composent. Dans un graphe pondéré, la longueur d'un chemin est la somme des poids des arêtes qui le composent.
Dans le graphe de la figure 2, la longueur du chemin $D \to A \to B \to C \to E$ est $1 + 3 + 12 + 8 = 24$.

3 Arbre couvrant de poids minimum

Un arbre couvrant de poids minimum d'un graphe pondéré est un arbre couvrant dont la somme des poids des arêtes est plus petite ou égale à la somme des poids des arêtes de tout autre arbre couvrant du graphe. Les algorithmes de recherche produisent des arbres couvrants qui ne sont pas nécessairement de poids minimum.

3.1 Algorithme de Kruskal

Cet algorithme permet de trouver l'arbre couvrant de poids minimum d'un graphe connexe et non orienté.
Les étapes de l'algorithme de Kruskal sont les suivantes :

1. Créer une liste contenant toutes les arêtes du graphe.
2. Trier les arêtes du graphe par ordre croissant de poids.
3. Sélectionner la première arête de la liste l'ajouter à l'arbre couvrant si elle ne crée pas de cycle, sinon la supprimer.
4. Répéter l'étape 2 jusqu'à ce que l'arbre couvrant soit complet.

Voici l'application de l'algorithme de Kruskal sur le graphe de la figure 2:

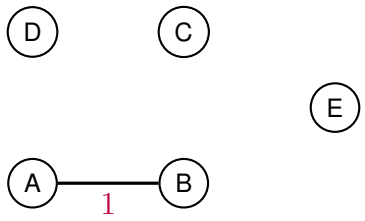


Figure 3: Sélection de la première arête : $(A, B, 1)$

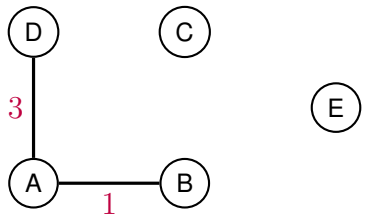


Figure 4: Sélection de la deuxième arête : $(A, D, 3)$

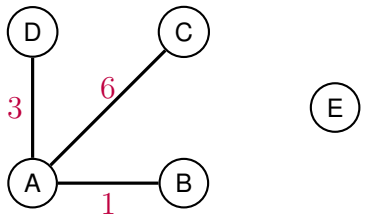


Figure 5: Sélection de la troisième arête : $(A, C, 6)$

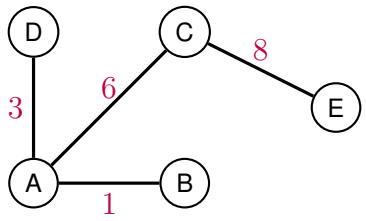


Figure 6: Sélection de la quatrième arête : $(C, E, 8)$.

4 Rappel - Les files et les piles

Une file est une structure de données qui permet de stocker des éléments dans un ordre précis. Les éléments sont ajoutés à la fin de la file et retirés du début de la file (comme dans une file d'attente). On parle de file FIFO (First In, First Out).
Dans une pile, les éléments sont ajoutés et retirés du même côté. On parle de pile LIFO (Last In, First Out).

En Python, on peut utiliser la classe `deque` du module `collections` pour obtenir une file ou une pile.
Voilà un exemple d'utilisation de la classe `deque`:

```python
from collections import deque

ma_file = deque () # Création d'une file
ma_file.append( 1 ) # Ajout d'un élément
ma_file.extend ([ 2 , 3 , 4 ]) # Ajout de plusieurs éléments
print(ma_file.popleft()) # 1
print(ma_file.pop()) # 4
```

Dans un parcours de graphe, en utilisant `popleft`, on obtient un parcours en largeur. En utilisant `pop`, on obtient un parcours en profondeur.

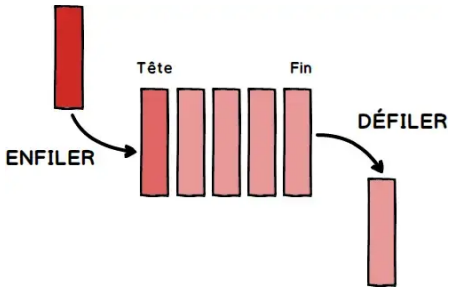


Figure 7: Dans une file, les éléments sont ajoutés et retirés à des extrémités différentes

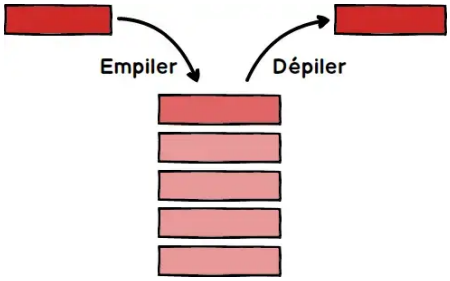


Figure 8: Dans une pile, les éléments sont ajoutés et retirés à la même extrémité

Figure 9: File et pile

5 File de priorité

Une file de priorité est une structure de données qui permet de stocker des éléments en leur associant une priorité. Les éléments sont retirés de la file en fonction de leur priorité.
En Python, on peut utiliser le module `PriorityQueue` du module `queue` pour obtenir une file de priorité.
Voilà un exemple d'utilisation de la classe `PriorityQueue`:

```python
from queue import PriorityQueue

ma_file = PriorityQueue() # Création d'une file de priorité

ma_file.put(( 1 , "A" )) # Ajout d'un élément A de priorité 1
ma_file.put(( 3 , "B" )) # Ajout d'un élément B de priorité 3
ma_file.put(( 2 , "C" )) # Ajout d'un élément C de priorité 2

while not ma_file.empty():
  print(ma_file.get()) # (1, "A") (2, "C") (3, "B")
```

6 Le problème du plus court chemin dans un graphe

6.1 Définition

Le problème du plus court chemin dans un graphe est un problème qui consiste à trouver le chemin le plus court entre deux sommets d'un graphe.

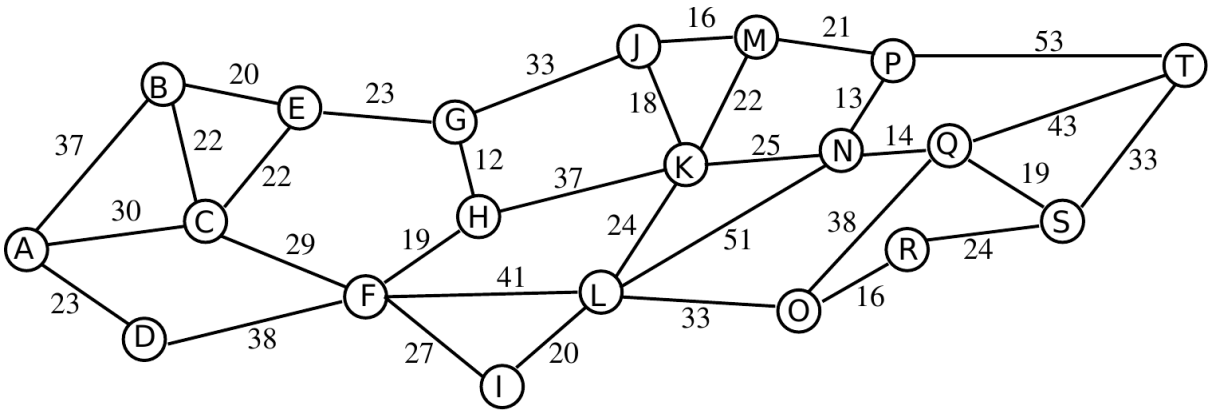


Figure 10: Trouver le plus court chemin devient de plus en plus difficile avec la taille du graphe

Quel est le plus court chemin entre A et T dans le graphe de la figure 10 ?

7 Algorithme de recherche de chemin

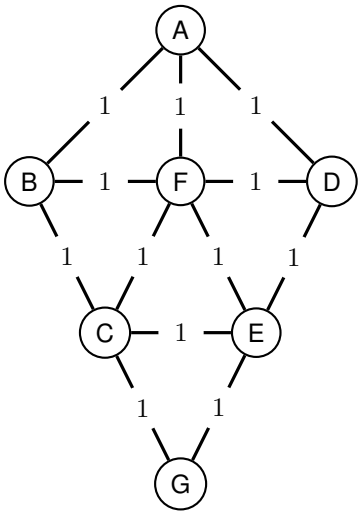
7.1 Algorithme de parcours en largeur

L'algorithme de parcours en largeur fonctionne en explorant en priorité les sommets les plus proches du sommet de départ.
Cet algorithme permet de trouver le plus court chemin dans mais uniquement dans les graphes non pondérés, comme les labyrinthes.
Voilà une implémentation de l'algorithme de parcours en largeur en Python qui utilise `deque`:

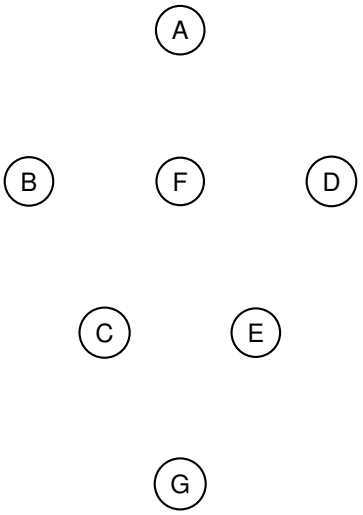
```
1  def parcours_largeur(graphe, depart):
2    visite = set()
3    file = deque([depart])
4    visite.add(depart)
5    while file:
6      sommet = file.popleft()
7      for voisin in graphe[sommet]:
8        if voisin not in visite:
9          visite.add(voisin)
10         file.append(voisin)
```

En remplaçant `file.popleft()` par `file.pop()` à la ligne 6 on obtient un parcours en profondeur.

A quel condition l'algorithme de parcours en largeur est optimal pour les graphes pondérés ?



Destination	Départ	Distance



7.2 Algorithme de Dijkstra

Cet algorithme est basé sur le principe de la recherche en largeur. Il fonctionne en explorant en priorité les sommets les plus proches du sommet de départ.

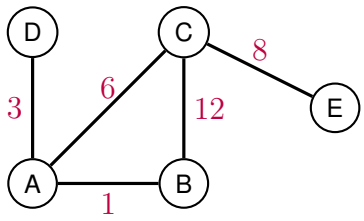


Figure 11: Dans ce graphe, le sommet de départ est A

Voilà une implémentation de l'algorithme de Dijkstra en Python, utilisant le module `PriorityQueue`:

```python
from queue import PriorityQueue
def dijkstra(graphe, depart):
  file = PriorityQueue()
  file.put((0, depart))
  visite = set()
  distances = {depart: 0}
  while not file.empty():
    (distance, sommet) = file.get()
    if sommet in visite:
      continue
    visite.add(sommet)
    for voisin, poids in graphe[sommet]:
      if voisin not in visite:
        nouvelle_distance = distances[sommet] + poids
        if voisin not in distances or nouvelle_distance < distances[voisin]:
          distances[voisin] = nouvelle_distance
          file.put((nouvelle_distance, voisin))
  return distances

graph = {
  'A': [('B', 1), ('C', 6), ('D', 3)],
  'B': [('A', 1), ('C', 12)],
  'C': [('A', 6), ('B', 12), ('E', 8)],
  'D': [('A', 3)],
  'E': [('C', 8)]
}
print(dijkstra(graph, 'A')) # {'A': 0, 'B': 1, 'D': 3, 'C': 6, 'E': 14}
```

Il est important de noter :

1. Que nous n'avons pas défini de sommet d'arrivée, l'algorithme retourne donc les distances minimales entre le sommet de départ et tous les autres sommets.
2. Que ce code ne retourne que les distances et non les chemins associés.

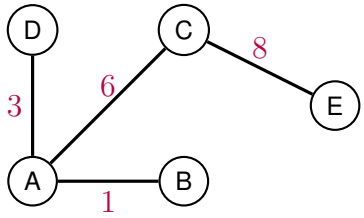
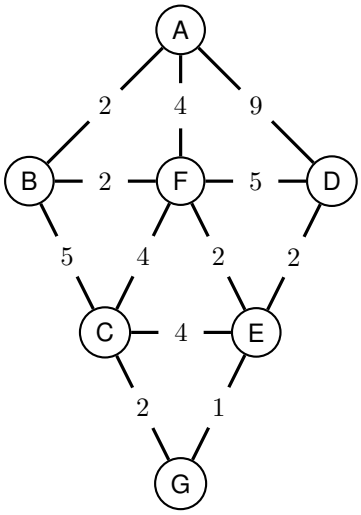


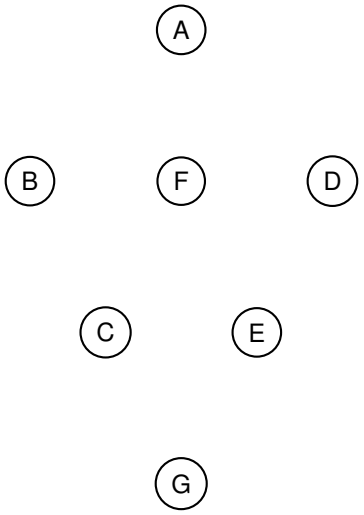
Figure 12: L'arbre couvrant correspondant aux distances trouvées

L'arbre couvrant associé aux distances trouvées est visible dans la figure 12.

! En empruntant les arêtes de cet arbre, on obtient le chemin le plus court entre le sommet de départ et les autres sommets.
En fonction du point de départ, l'arbre couvrant peut être différent.



Destination	Départ	Distance



7.3 Approche heuristique

Vous avez déjà vu l'algorithme A^* pour rechercher efficacement un chemin dans un labyrinthe.
Formellement l'algorithme A^* va choisir le sommet qui minimise la fonction $f(n) = g(n) + h(n)$ où :

- $g(n)$ est le coût du chemin depuis le sommet de départ jusqu'au sommet n.
- $h(n)$ est une heuristique qui estime le coût du chemin depuis le sommet n jusqu'au sommet d'arrivée.

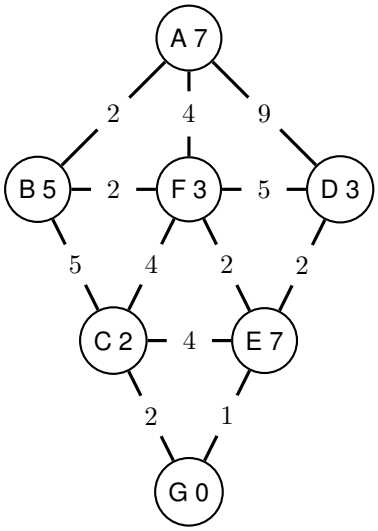
Dans les labyrinthes :

- $g(n)$ correspondait à la longuer du chemin depuis le sommet de départ jusqu'au sommet n, i.e. le nombre d'arêtes empruntées jusqu'à n.
- $h(n)$ correspondait à la distance de Manhattan entre le sommet n et le sommet d'arrivée.
- La recherche en largeur et Dijkstra produiront des résultats similaires et optimaux.

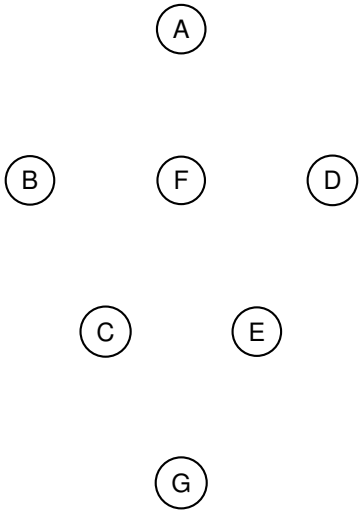
L'approche de Dijkstra est un cas particulier de l'algorithme A^* où $h(n) = 0$ pour tout sommet n.

! L'optimalité et l'efficacité de l'algorithme A^* dépendent de la qualité de l'heuristique $h(n)$.
Une heuristique admissible est une heuristique qui ne surestime pas le coût du chemin restant, dans ce cas, l'algorithme A^* est optimal.

Si l'heuristique surestime le coût du chemin restant, la garantie de l'optimalité de l'algorithme A^* n'est plus assurée. Si l'heuristique sous-estime le coût du chemin restant, l'algorithme A^* est toujours optimal, mais il sera moins rapide.



Destination	Départ	Distance



7.4 Points à retenir

- Un graphe pondéré est un graphe où des valeurs numériques sont assignées aux arêtes.
- La longueur d'un chemin dans un graphe pondéré est la somme des poids des arêtes qui le composent.
- Le problème du plus court chemin dans un graphe est un problème qui consiste à trouver le chemin le plus court entre deux sommets d'un graphe.
- L'algorithme de parcours en largeur fonctionne en explorant en priorité les sommets les plus proches du sommet de départ.
- L'algorithme de Dijkstra est basé sur le principe de la recherche en largeur. Il fonctionne en explorant en priorité les sommets les plus proches du sommet de départ.
- L'algorithme A^* est une approche heuristique pour résoudre le problème du plus court chemin dans un graphe. Cet algorithme est également basé sur le principe de la recherche en largeur.

8 Exercices papier

Dans tous les exercices, A sera le sommet de départ et J le sommet d'arrivée.
Utiliser une recherche en largeur pour trouver le plus court chemin entre A et J dans l'exercice 1, l'algorithme de Dijkstra pour l'exercice 2 et l'approche heuristique pour l'exercice 3 et 4 (Le nombre à côté du sommet correspond à la valeur de notre heuristique : une estimation de la distance depuis ce point jusqu'au point d'arrivée J).

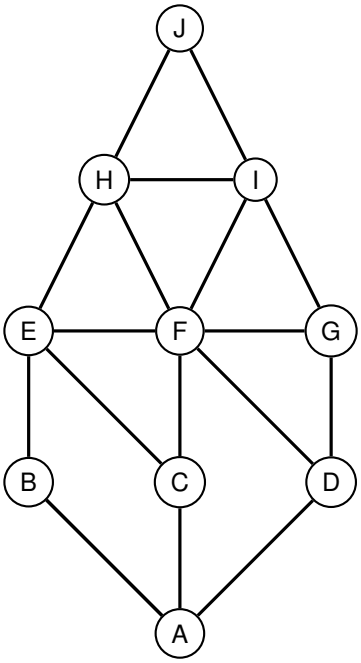


Figure 13: Exercice 1

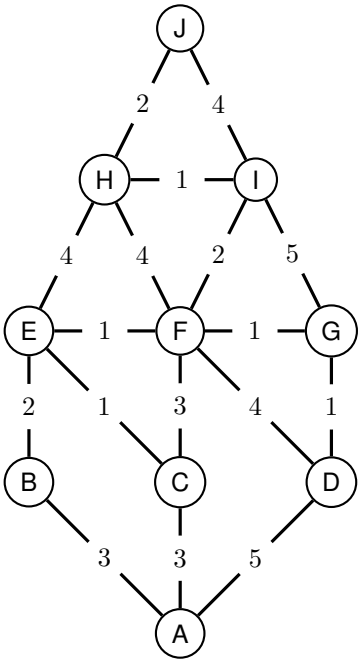


Figure 14: Exercice 2

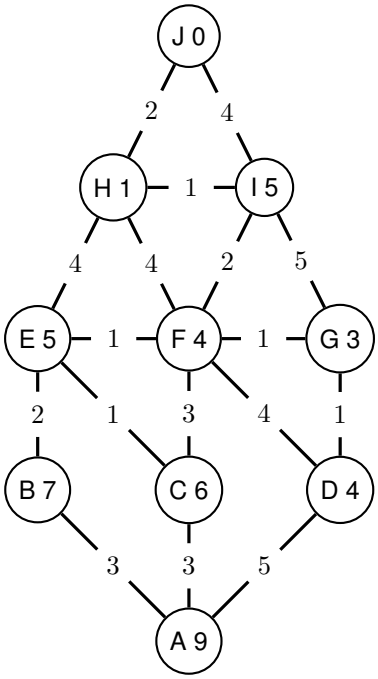


Figure 15: Exercice 3

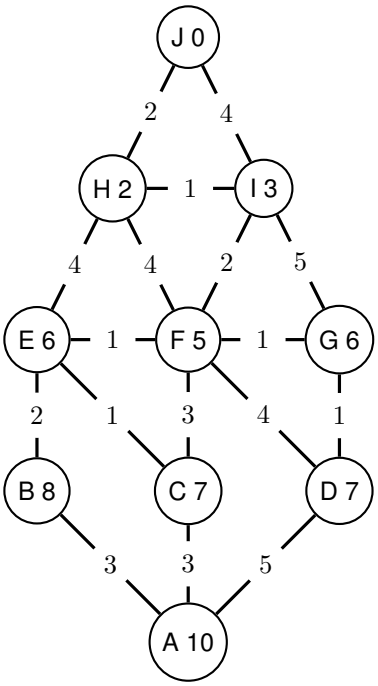


Figure 16: Exercice 4