ITC – cours n°15
Problème du plus court chemin

I. Position du problème

Dans un graphe pondéré (orienté ou non), on définit la longueur d'un chemin $c = \{u_0, u_1, u_2, \dots u_{n-1}, u_n\}$ entre deux sommets u_0 et $u_n = u_f$ comme la somme des poids des arcs qui constituent le chemin :

$$w(c) = \sum_{i=0}^{n-1} w_{(u_i, u_{i+1})}$$

et un plus court chemin est un chemin tel que

$$\begin{cases} w(c) = \min\{w(c'), c' = \{u_0, \dots, u_f\}\} & \text{si } u_0 \to u_f \\ \infty & \text{sinon} \end{cases}$$

Dans ce cours, nous restreindrons notre étude au **problème de recherche du plus court chemin à origine unique** : étant donné un graphe $G = (S, A)$, on souhaite trouver un plus court chemin depuis un sommet origine donné $u_0 \in S$ vers n'importe quel sommet $v \in S$.

Ce problème trouve de nombreuses applications d'optimisation : l'idée la plus intuitive est celle de limiter la distance d'un point à un autre si le graphe représente un réseau routier, mais on peut souhaiter minimiser de nombreuses grandeurs qui s'accumulent le long d'un chemin : un temps, un coût, des pénalités...Nous avons déjà vu que le parcours en largeur répond à cette problématique dans le cas d'un graphe non pondéré. Pour adapter cela à un graphe pondéré, on ajoute la prise en compte de la longueur du chemin.

Beaucoup d'autres problèmes peuvent être résolus par l'algorithme à origine unique, notamment les variantes suivantes :

- **plus court chemin à destination unique** : trouver un plus court chemin vers un sommet de destination u_f à partir de n'importe quel sommet v. En inversant le sens de chaque arc du graphe, on peut ramener ce problème à un problème à origine unique ;
- **plus court chemin pour un couple de sommets donné** : trouver un plus court chemin entre deux sommets donnés u_0 et u_f. Si on résout le problème à origine unique pour le sommet origine u, on résout ce problème également. Par ailleurs, on ne connaît aucun algorithme qui soit meilleur asymptotiquement que les meilleurs algorithmes à origine unique dans le pire des cas ; nous verrons en revanche un algorithme qui permet d'obtenir plus rapidement une solution dans des cas adaptés ;
- **plus court chemin pour tout couple de sommets** : trouver un plus court chemin de u à v pour tout couple de sommets u et v. Ce problème peut être résolu en exécutant un algorithme à origine unique à partir de chaque sommet ; mais on peut généralement le résoudre plus rapidement. Ces améliorations dépassent le cadre de ce cours.

II. Propriété des plus courts chemins et relâchement d'un arc

Pour « améliorer » le parcours en largeur, il faut en reprendre le principe en gérant un majorant de la distance entre le sommet de départ et chaque autre sommet. Ce majorant est initialisé à ∞ et est abaissé au fur et à mesure de l'avancée du parcours (il reste à ∞ si le sommet de départ et le sommet v considéré n'appartiennent pas à la même composante connexe).

On s'appuie sur une propriété assez intuitive des graphes pondérés : les sous-chemins d'un plus court chemin sont eux-mêmes des plus courts chemins. Prenons l'exemple du graphe ci-dessous : le plus court chemin de A à D est $A - B - E - D$, et le plus court chemin de A à E (sous-chemin de $A - B - E - D$) est lui-même $A - B - E$; de même, le plus court chemin de A à B est $A - B$.

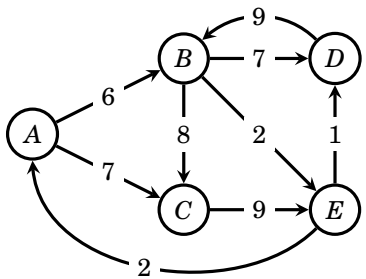


Figure 1 : Exemple de graphe pondéré

En s'appuyant sur cette propriété, on peut proposer une procédure de relâchement d'un arc (u, v) : connaissant un majorant de la distance entre le sommet origine u_0 et le sommet v, on regarde si on peut abaisser ce majorant en empruntant l'arc (u, v) ; si oui, le sommet précédent v devient u. On montre ci-dessous deux exemples de relâchement d'arc, avec ou sans mise à jour des valeurs selon le chemin précédemment trouvé vers v (on note sur chaque sommet son précédent et la distance actuelle au sommet origine u_0) :

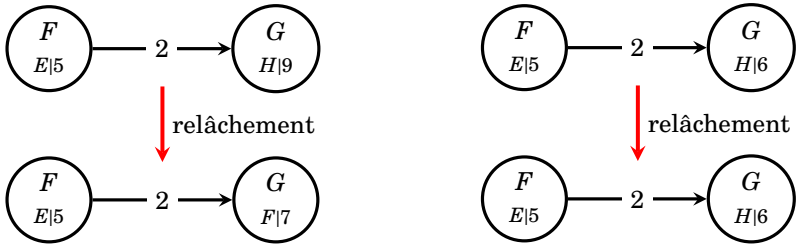


Figure 2 : Relaâchement d'un arc

III. Algorithme de Dijkstra

3.1. Principe

L'algorithme de Dijkstra s'appuie sur le parcours en largeur et la technique du relâchement d'un arc :

- on choisit le prochain sommet u à visiter en prenant celui qui a la plus petite distance depuis la source : on ne trouvera pas de meilleur chemin pour lui ;
- on visite u en relâchant tous les arcs (u, v) partant de ce sommet et atteignant les sommets v non visités.

Par rapport au parcours en largeur, on remplace donc la file dans laquelle on place les sommets en attente d'être visités par une **file de priorité** (nous discuterons plus loin des détails de cette opération). Une fois le sommet le plus proche extrait de la liste et ses arcs relâchés, on ne reviendra plus sur le traitement de ce sommet : il s'agit donc d'un algorithme glouton.

On présente une application pas à pas de l'algorithme en partant du sommet A sur la figure 3

3.2. Implémentation

Une implémentation possible serait alors la suivante ; on utilise ici comme liste de priorité la liste des sommets dont on extrait le sommet de plus petite distance par un parcours en $\mathcal{O}(n_S)$.

```
def relâcher(g, u, v, distance, précédent):
    if distance[u] + g[u][v] < distance[v]:
        distance[v] = distance[u] + g[u][v]
        précédent[v] = [u]

def extraire_min(distance, état):
    """
    Extrait de façon naïve le sommet non visité le plus proche du départ.
    S'il ne reste aucun sommet non visité, on renvoie ''
    """
    u_min, mini = '', float('inf')
    for u in distance.keys():
        if état[u] == 'V':
            if distance[u] < mini:  # on cherche un minimum
                u_min, mini = u, distance[u]
    return u_min

def visiter_dijkstra(g, u, précédent, état, distance):
    for v in g[u]:
        if état[v] != 'R':
            relâcher(g, u, v, précédent, distance)
    état[u] = 'R'
```

```
def dijkstra(g, u0):
    précédent, état, distance = init_parcours(g)
    distance[u0] = 0
    u = u0
    while u != '':                                  # répété au plus ns fois
        visiter_dijkstra(g, u, précédent, état, distance)  # O(deg(u))
        u = extraire_min(état, distance)                   # O(ns)
  return # à adapter selon ce que l'on souhaite
```

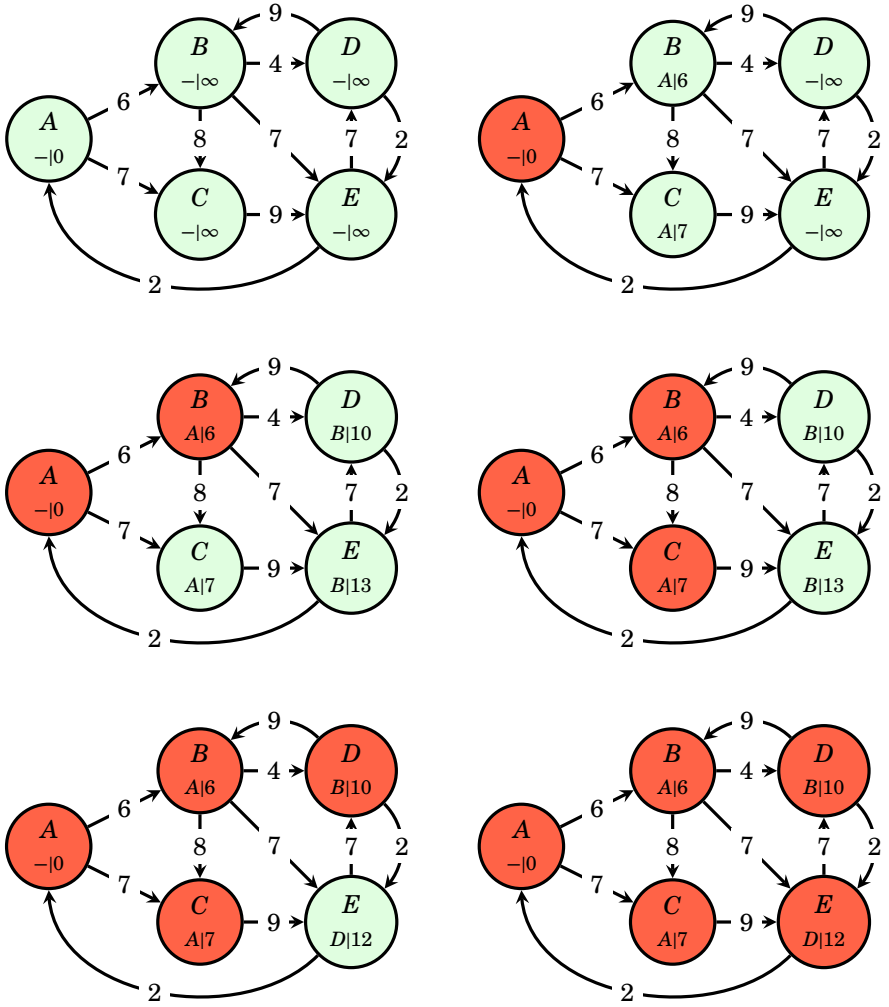


Figure 3 : Exemple d'application de Dijkstra

3.3. Complexité

L'analyse de la complexité est la suivante :

- grâce au système de coloriage, chaque sommet est visité au plus une fois, donc il y a n_S appels appels à `visiter_dijkstra` et à `extraire_min` (exactement n_S appels si le graphe est connexe) ;
- dans `visiter_dijkstra` , chaque arc partant de u est relâché une seule fois, donc dans **l'ensemble** des appels à `visiter_dijkstra` chaque arc du graphe est relâché au plus une fois (exactement une fois si le graphe est connexe), soit au total kn_A opérations ;
- `extraire_min` passe en revue tous les sommets, donc **chaque appel** coûte $k'n_S$;

ce qui conduit finalement à une complexité

$$k'n_S^2 + kn_A = \mathcal{O}(n_S^2 + n_A) = \mathcal{O}(n_S^2)$$

puisqu'il y a au maximum $\mathcal{O}(n_S^2)$ arcs.

On aurait pu gérer la file de priorité comme une liste Python (ou une liste chaînée) que l'on garde triée selon les distances : en tenant compte du fait qu'à chaque visite d'arc la liste reste presque triée, on peut espérer la retrier en $\mathcal{O}(n_S)$ avec un tri par insertion.

On peut implémenter une meilleure version si le graphe est peu dense ($n_A \ll n_S$) en implémentant la file de priorité de façon plus efficace. Une structure de données appelée **tas binaire** qui permet de gérer la file de priorité, dans laquelle extraire le minimum est en $\mathcal{O}(\log n_S)$ et mettre à jour l'ordre de la file de priorité lors du relâchement des arcs est en $\mathcal{O}(\log n_S)$. Dans ce cas, la complexité diminue puisque

- les n_A relâchements d'arc coûtent maintenant $kn_A \log n_S$;
- les n_S extraction du minimum coûtent maintenant $k'n_S \log n_S$

soit une compexité totale

$$\mathcal{O}((n_S + n_A)\log n_S)$$

ce qui est une amélioration dès lors que $n_A \ll n_S^2/\log n_S$. La file de priorité « naïve » que nous avons utilisée précédemment peut néanmoins s'avérer plus efficace si le graphe est trop dense, les préfacteurs compensant le gain de complexité.

Ces structures de données ne sont pas au programme, mais nous présentons leur principe dans la section suivante.

3.4. (Hors-progamme) Structure de tas pour les files de priorité

Un tas binaire min est un arbre enraciné dont chaque sommet possède au plus deux enfants, appelés enfant gauche et droite, tel que :

- le parent a toujours une valeur plus petite que chacun des enfants ;
- l'arbre est parfaitement équilibré : tous les niveaux depuis la racine sont entièrement remplis sauf le dernier, sa hauteur est donc $\log_2(n)$.

Un exemple se trouve sur la figure 4.

On peut également créer des tas-max, où l'ordre est inversé, selon l'usage que l'on veut faire de la structure.

> **i** Dans la pratique, on peut utiliser un tableau pour stocker l'arbre, avec pour un sommet à l'indice i, l'enfant gauche à l'indice $2i$, l'enfant droit à l'indice $2i+1$ et le parent à l'indice $i//2$. Des implémentations existent déjà en Python, par exemple la classe `heapq` .

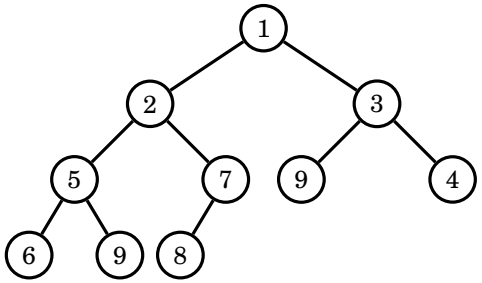


Figure 4 : Un tas-min

On peut mettre à jour le tas tout en conservant cette propriété si on place en haut un élément qui n'est pas plus petit que ses enfants, en le permutant avec le plus petit de ses enfants de façon à le faire redescendre d'un étage, puis en répétant jusqu'à ce qu'il soit à la bonne place. Cette procédure, appelée entasser l'élément, permet d'insérer de nouveaux éléments dans le tas sans en perdre la propriété. On peut voir un exemple d'exécution sur la figure 5. Puisqu'il y a au plus des permutations depuis la racine jusqu'en bas de l'arbre, la procédure est en $\mathcal{O}(\log n)$.

On comprend alors qu'on peut aisément récupérer le plus petit élément :

- on enlève la racine, qui est donc le plus petit élément ($\mathcal{O}(1)$) ; il sera renvoyé à la fin de la fonction ;
- on met à la place de la racine le dernier élément en bas du tas, afin de retrouver un arbre bien défini pour lequel la propriété de tas est respectée sauf à la racine ($\mathcal{O}(1)$) ;
- on entasse l'élément à la racine afin de retrouver la prorpiété de tas ($\mathcal{O}(\log n)$).

Extraire le plus petit élément d'une file de priorité gérée avec un tas-min est donc en $\mathcal{O}(\log n)$.

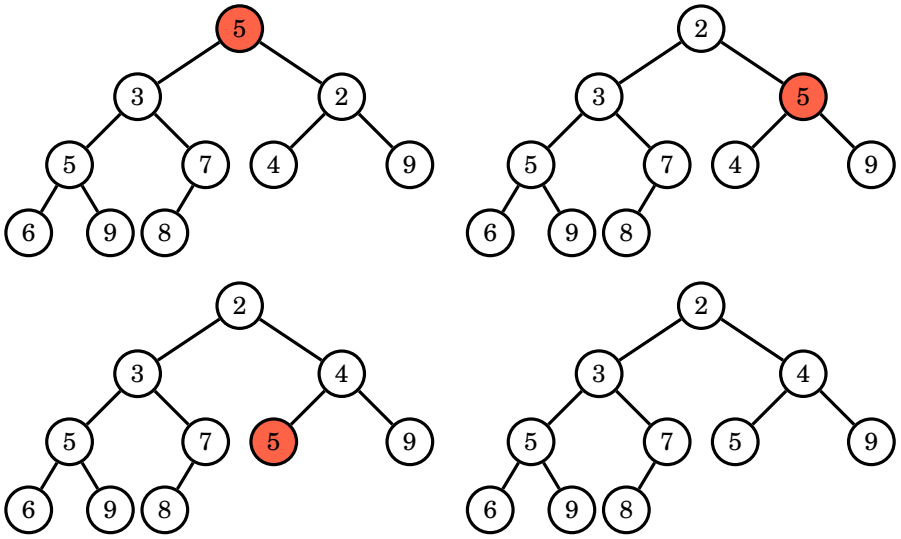


Figure 5 : Positionnement d'un élément

De même, on peut diminuer les valeurs dans un tas en le permutant avec son parent jusqu'à le remettre en bonne position. On peut voir un exemple figure 6, et on comprend que comme l'entassement, cette opération est en $\mathcal{O}(\log n)$.

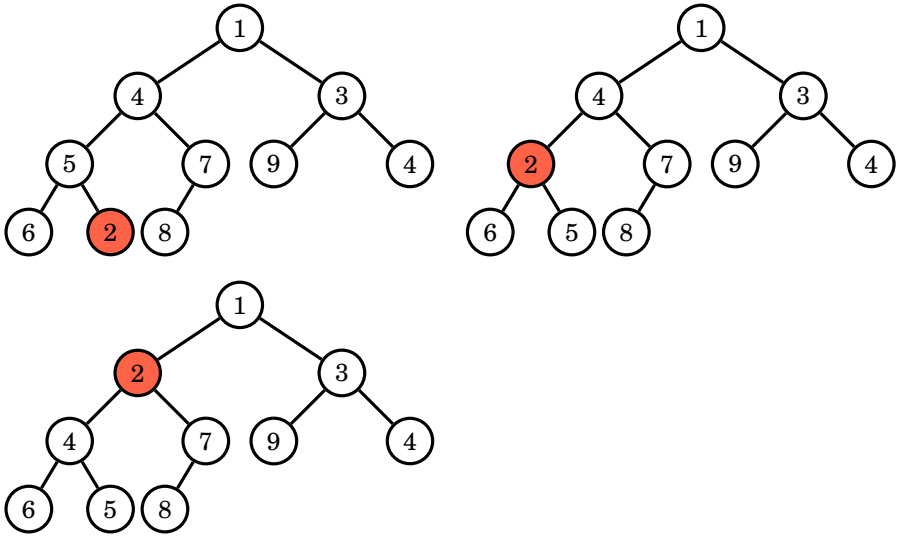


Figure 6 : Augmentation de la priorité d'un élément

IV. Algorithme « A-étoile » (ou « A-star »)

4.1. Idée générale

Le problème de trouver un plus court chemin d'un sommet u_0 à un **unique sommet d'arrivée** u_f est légèrement différent : l'algorithme de Dijkstra répond à cette problématique (il suffit de s'arrêter dès que le sommet u_f est extrait de la file de priorité), mais n'est pas forcément assez rapide : en effet, on peut explorer de proche en proche des chemins qui sont certes les plus courts pour aller à certains sommets u, mais qui sont inutiles pour trouver un chemin jusqu'à u_f.

L'algorithme A*reprend l'idée de Dijkstra et ajoute une pénalité aux sommets qui sont moins susceptibles d'amener vers le sommet d'arrivée u_f ; ainsi, on explorera en priorité les sommets les plus susceptibles de nous amener dans la bonne direction, et on peut ainsi espérer gagner du temps sur l'exploration du graphe en explorant uniquement les parties « intéressantes ».

La difficulté est donc de trouver un moyen de sélectionner les sommets probablement plus intéressants. Une telle fonction, destinée à favoriser certains sommets au détriment d'autres, est ici appelée une **heuristique**. Celle-ci dépend donc du problème considéré.

4.2. Un cas concret

L'algorithme A*dépend de notre capacité à produire une heuristique ; il est donc particulièrement utilisé dans les problèmes de recherche de chemin géométrique, dans lequel on peut définir une distance entre un sommet en cours d'exploration et le sommet d'arrivée : plus la distance est courte, plus l'heuristique donnera une pénalité basse[1].

Dans ce paragraphe, nous nous déplacerons sur un damier (représentant l'espace 2D ou un labyrinthe), ce qui simplifie les calculs, avec des arêtes si on peut se déplacer entre deux sommets adjacents et une absence d'arête s'il y a un obstacle : on utilisera comme heuristique la distance de Manhattan, c'est-à-dire qu'entre deux sommets u de coordonnées (i_u, j_u)et v de coordonnées (i_v, j_v),

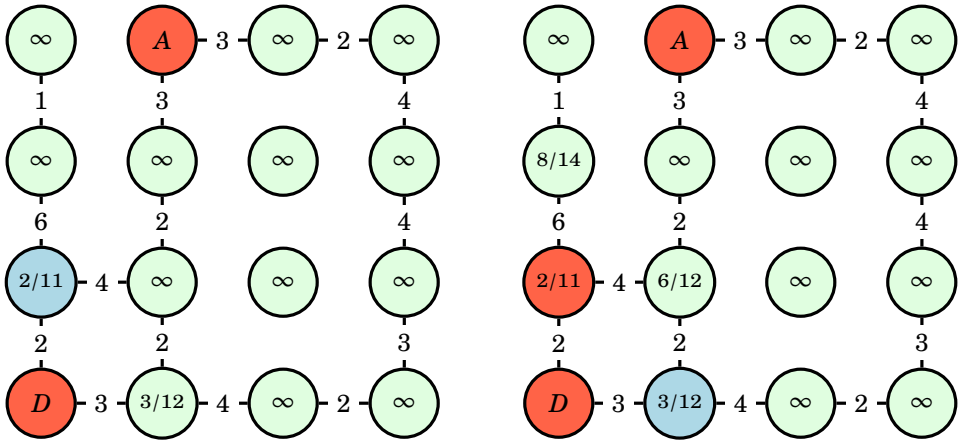
$$d_{u,v} = |i_u - i_v| + |j_u - j_v|$$

Cette distance correspond au nombre de déplacements à effectuer sur le damier pour aller de u à v ; si le graphe est pondéré, on multiplie par le poids moyen pour trouver une estimation du coût restant à partir du sommet u pour aller au sommet v, et on en déduit le coût estimé jusqu'au sommet d'arrivée ; c'est cette estimation qui sert pour la file de priorité.

> Si on peut se déplacer en diagonale (s'il y a des arêtes donc), la distance euclidienne $\sqrt{(i_u - i_v)^2 + (j_u - j_v)^2}$ est plus adaptée.

[1] Une des principales applications de A*est d'ailleurs le pathfinding dans les jeux vidéos : on peut définir une heuristique à partir d'une distance géométrique, et on préfère un algorithme qui répond rapidement même si la solution n'est pas optimale.

Un exemple d'exploration est proposée en figure 7, avec un graphe de poids moyen 3 : on représente le coût pour aller jusqu'à un sommet ainsi que le coût total estimé pour aller à l'arrivée.



On explore le sommet de CE 11 — On explore un sommet de CE 12

On explore le sommet de CE 11 — On explore le sommet de CE 10

Figure 7 : Exemple dans un labyrinthe

4.3. Implémentation

On retrouve les principes généraux de visite d'un sommet et de relâchement d'un arc, adaptés avec l'estimation du coût restant du chemin. On propose ci-dessous une version adaptée à un graphe non pondéré pour éviter d'alourdir le calcul.

```python
def extraire_min_heur(état, dist, ce):
    """
    Extrait de façon naïve le sommet non visité le plus prometteur, donc
    de (dist + ce) minimal.
    S'il ne reste aucun sommet accessible non visité, on renvoie ''
    """
    u_min, mini = '', float('inf')
    for u in dist.keys():
        if état[u] == 'V':
            if dist[u] + ce[u] < mini:  # on cherche un minimum
                u_min, mini = u, dist[u] + ce[u]
    return u_min


def astar(g, fonction_heuristique, u1, u2):
    # estimation de la distance restante entre chaque sommet et u2
    ce = {u: fonction_heuristique(u, u2) for u in g.keys()}
    # initialisation du parcours
    précédent, état, distances = init_parcours(g)
    état[u1], distances[u1] = 'B', 0
    u = u1
    # Début de l'algorithme
    while u != '' and u != u2:  # répété au plus ns fois
        visiter_dijkstra(g, u, précédent, état, distances)
        u = extraire_min_heur(état, distances, ce)  # O(ns)
    return # à adapter selon ce que l'on souhaite
```

On note les différences principales par rapport à Dijkstra :

- on garde en mémoire à la fois la distance jusqu'au sommet et le coût estimé de la fin à partir de ce sommet ;
- `extraire_min` suit toujours le même principe mais agit sur le coût estimé ;
- on s'arrête dès que l'on trouve le sommet d'arrivée.

La fonction heuristique doit être fournie en entrée.