7| Bases des graphes

1 Définition d'un graphe

L'utilisation de graphes en informatique est souvent très pratique pour résoudre certains problèmes (carte de points, chemin dans un labyrinthe, trajets par GPS, compression de données, etc.). Souvent, cette résolution passe ensuite par l'utilisation d'une pile gérant l'ensemble des possibilités de parcours du graphe.

1.1 Graphes non orientés

Un **graphe** $G = (V, E)$ est défini par l'ensemble fini $V = \{v_1, v_2, \ldots, v_n\}$ dont les éléments sont appelés **sommets** (Vertices), et par l'ensemble fini $E = \{e_1, e_2, \ldots, e_m\}$ dont les éléments sont appelés **arêtes** (Edges).

Si l'arête e relie les sommets a et b, on dira que ces sommets sont **adjacents** et que l'arête est **incidente** aux sommets qu'elle relie.

Le nombre de sommets du graphe s'appelle l'**ordre** du graphe et le **degré d'un sommet** v, noté d(v) est le nombre d'arêtes qui lui sont incidentes.

Un **chemin** de longueur p dans le graphe est une suite de p+1 sommets. Si le premier sommet est identique au dernier dans un chemin, on parle alors de **cycle** et un graphe est dit **acyclique** s'il ne possède pas de cycle.

Un graphe non orienté $G = (V, E)$ est dit **connexe** si, pour deux sommets quelconques u et v de V, il existe un chemin de u à v dans G.

Un **arbre** est un type de graphe connexe acyclique dans lequel les sommets peuvent être distingués en 3 types :

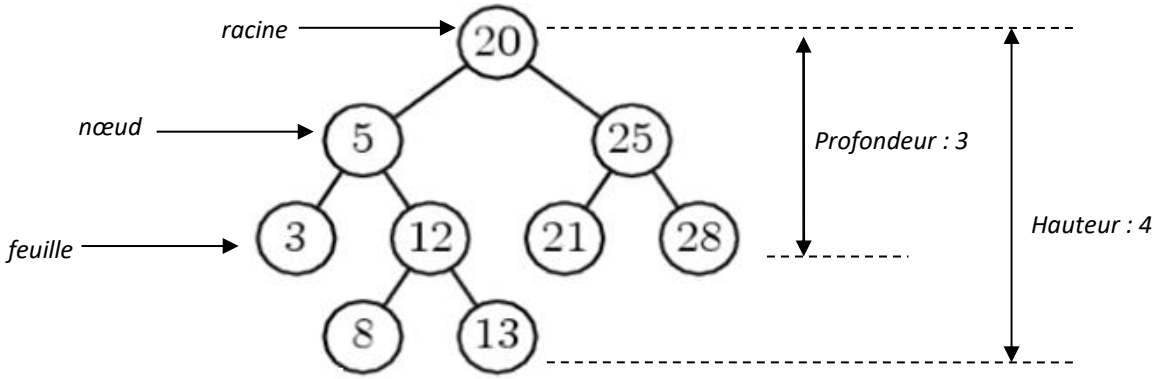
- Un sommet qui possède des fils et un père est un **nœud** ;
- Un sommet qui possède un père mais aucun fils est une **feuille** ;
- Un sommet qui possède des fils mais aucun père est une **racine**.

On peut alors définir :

- La **profondeur** d'un nœud : distance (c'est-à-dire le nombre de sommets) séparant un sommet de la racine ;
- La **hauteur** d'un arbre : nombre de sommet traversés pour aller de la racine à la feuille la plus profonde ;
- La **taille** de l'arbre : nombre de nœuds et de feuilles.

Enfin, on a pour habitude de représenter l'arbre "à l'envers" en présentant sa racine en haut. Les sommets peuvent également être **étiquetés** (c'est-à-dire contenir une valeur) et les arêtes peuvent avoir un **poids**.

Exemple d'un arbre étiqueté non pondéré :



1.2 Graphes orientés

Un graphe est dit **orienté** si un sens de parcours des arêtes est défini. D'ailleurs, le terme d'arête sont est remplacé par **arc**.

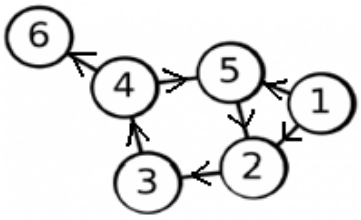
Exemple d'un graphe orienté (il existe une infinité de représentations possibles) :

Ensemble des sommets :

$$V = \{1,2,3,4,5,6\}$$

Ensemble des arcs :

$$E = \{(1,2), (1,5), (2,3), (3,4), (4,5), (4,6), (5,2)\}$$



Puisque les arcs ont une orientation, on ne parle plus du degré d'un sommet v, mais de son **degré entrant** (nombre d'arcs de la forme (u,v)) et de son **degré sortant** (nombre d'arcs de la forme (v,u)). On notera $d_+(v)$ et $d_-(v)$ les degrés sortant et entrant d'un sommet.

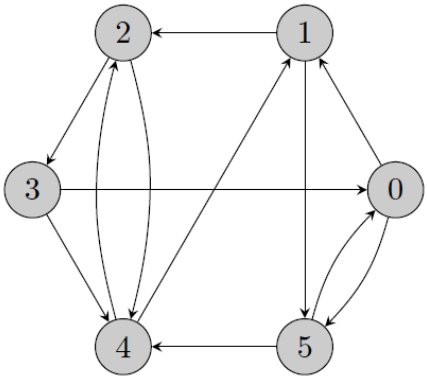
1.3 Implantations des graphes

Du point de vue des applications, on s'intéressera à des graphes « statiques », c'est-à-dire que l'ensemble des sommets est fixé. On supposera que l'ensemble des sommets est $[\![0, n-1]\!]$. Il y a essentiellement deux manières d'implémenter un graphe :

- via des **listes d'adjacences** : on stocke pour chaque sommet la liste de ses voisins. Cette représentation est économique en mémoire, et adaptée aux graphes ayant peu d'arcs/arêtes (on dit qu'ils sont « **creux** »). On pourrait également utiliser des dictionnaires à la place des listes.
- via une **matrice d'adjacences** : on stocke à la case (i,j) d'une matrice n x n la présence ou non d'un arc entre les sommets i et j. Cette représentation est bien adaptée aux graphes « **denses** » (ayant beaucoup d'arcs/arêtes).

Implémentation « creuse »

Pour un graphe à n sommets dont les sommets sont $[\![0, n-1]\!]$, on utilise une liste G de taille n. L'élément G[i] est une liste, contenant les voisins de i. En général, on n'impose pas que les voisins de i soient ordonnés dans G[i].



Par exemple, le graphe orienté de la figure précédente peut être implémenté comme suivant :

$$G = [[5,1],[2,5],[3,4],[0,4],[2,1],[0,4]]$$

Pour représenter un graphe non orienté, on l'implémente simplement comme un graphe orienté, en dupliquant l'information : si l'arête $\{i,j\}$ est présente, i se trouve dans la liste d'adjacence de j et réciproquement.

Cette représentation est économique, car elle nécessite un espace mémoire en $\boldsymbol{O}(|\boldsymbol{V}| + |\boldsymbol{E}|)$. Parcourir les voisins d'un sommet se fait en **temps linéaire** en son degré, par contre il faut parcourir une liste d'adjacence pour tester l'existence d'un arc ou une arête.

Implémentation « dense »

Une manière de représenter un graphe dont les sommets sont $[\![0, n-1]\!]$ est d'utiliser une matrice pour indiquer l'existence d'un arc : la **matrice d'adjacence**.

La matrice d'un adjacence d'un graphe $[\![0, n-1]\!]$ est la matrice $M = \left(m_{i,j}\right)_{0 \le i,j \le n-1}$ définie par :

$m_{i,j} = \begin{cases} 1 & si\ (i,j) \epsilon E \\ 0 & sinon \end{cases}$.

Par exemple, la matrice du graphe précédent est :

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Cette matrice peut être implémentée comme une **liste de listes** ou à l'aide de la bibliothèque numpy comme un **array** (matrice).

Cette représentation n'est pas très économique en mémoire pour les graphes ayant peu d'arcs, car elle nécessite toujours un espace en $\boldsymbol{O}(|\boldsymbol{V}|^2)$. Parcourir les voisins d'un sommet se fait en temps

$\boldsymbol{O}(|\boldsymbol{V}|)$ quel que soit le degré du sommet, par contre tester l'existence d'un arc ou une arête se fait en **temps linéaire**.

2 Les piles de données

Dans l'optique de parcourir des graphes, il nous faut définir deux structures abstraites très utilisées en informatique : la **pile** et la **file**. Le mot abstrait signifie que l'implémentation concrète joue un rôle secondaire, ce qui importe est la description de la structure et des opérations que l'on peut lui appliquer. Bien sûr, lorsqu'on souhaite réaliser une implémentation, on visera à avoir la meilleure complexité possible.

Historiquement, les **piles** et les **files** (ou **stack** et **queue** en anglais) de données sont, avec les tableaux non ordonnés, les premières structures utilisées en informatique. Simples à comprendre et à implémenter, ces structures se sont imposées naturellement dès les débuts de l'informatique. Les structures de piles sont utilisées dans de nombreux domaines de l'informatique : mémorisation des pages Web visitées dans un navigateur, fonction UNDO (Ctrl+Z/Y) d'un traitement de texte, algorithmes récursifs, etc…

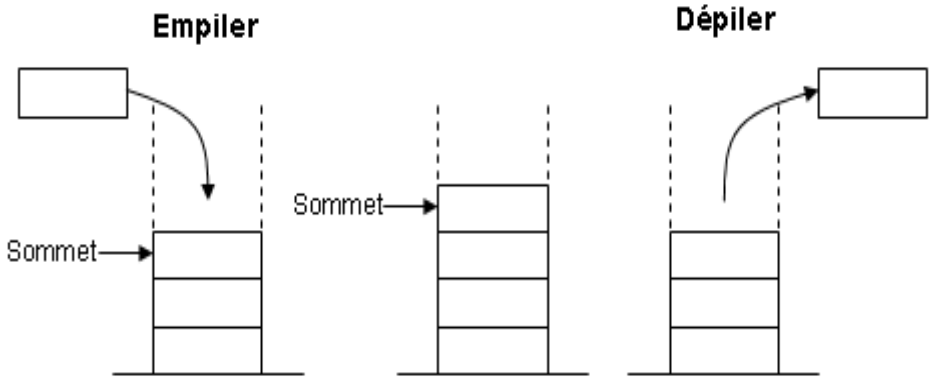
2.1 Définition

Les piles sont des structures de type Last In First Out (**LIFO**) : le dernier élément ajouté à une pile sera le premier à en être extrait. A la manière d'une pile d'assiettes, on ajoute les éléments « les uns par-dessus les autres ». On peut alors définir 2 actions principales sur la pile :

- **Empiler** une valeur (push en anglais) : c'est-à-dire l'insérer au sommet de la pile.
- **Dépiler** (pop en anglais) : c'est-à-dire supprimer la valeur au sommet.

Quelques opérations supplémentaires naturelles peuvent venir compléter les 2 premières :

- Connaître la hauteur de la pile, c'est-à-dire le nombre d'éléments qu'elle contient ;
- Vérifier si la pile est vide / vérifier si la pile est pleine ;
- Vider entièrement la pile.



La structure abstraite de **file** est semblable à celle d'une pile, mais suit le principe First In First Out (**FIFO**) : le premier élément ajouté à une pile sera le premier à en être extrait.

Dans l'absolu, les piles et les files peuvent être bornées (elles ont une taille finie et donc une capacité qu'on ne peut dépasser) ou non (leur taille est « à priori » illimitée).

2.2 Implémentation

Les piles bornées sont des piles dont la taille est fixée à la création. Une liste p de taille n suffisamment grande est utilisée pour cela. Le premier élément p[0] contient le nombre d'éléments contenus dans la pile. Le dernier élément empilé est en $p[0]^{éme}$ position ; ainsi pour le récupérer, on fait appel à p[p[0]].

Voici quelques exemples de scripts permettant de construire et de réaliser des opérations élémentaires sur les piles :

```
>>> def creePile(n) :
>>>     res = [None]*(n+1)      # +1 car il faut une case pour la taille
>>>     res[0] = 0              # initialement la pile est vide
>>>     return res
```

```
>>>def empile(x,p) :
>>>     p[0] = p[0] + 1
>>>     p[p[0]] = x
```

```
>>>def depile(p) :
>>>     p[0] = p[0] – 1         # Il n'est pas nécessaire de réellement effacer l'élément
>>>     return p[p[0]+1]
```

```
>>>def taille(p) :
>>>     return p[0]
```

```
>>>def sommet(p) :
>>>     return p[p[0]]
```

Remarque : La pile étant représentée par une liste (ou array) on peut utiliser plusieurs fonctions natives afin de remplacer efficacement les scripts ci-dessus. **L.append(element)** pour ajouter un élément et **L.pop()** pour supprimer le dernier élément et renvoyer sa valeur. Ces fonctions permettent de travailler avec des piles de tailles variables.

Par ailleurs, le module **collections** propose notamment la structure **deque**, pour double ended queue, c'est-à-dire en français file à deux bouts. Concrètement, c'est une structure de stockage linéaire des éléments (comme une liste !), mais qui fournit la possibilité d'effectuer l'insertion ou la suppression des deux côtés en temps constant.

La création se fait via **deque()**, et les opérations d'ajout sont **append** (à droite) et **appendleft** (à gauche). De même pour la suppression avec **pop** et **popleft**. On peut tester si une file à deux bouts est vide en regardant sa longueur.

```
>>> from collections import deque                # importation
>>> f = deque()                    # une file à deux bouts vide
>>> for i in range(5):
>>>     f.append(i)                # ajout à droite
>>> f
deque([0, 1, 2, 3, 4])
>>> f.pop()                # suppression à droite
4
>>> f.appendleft(5)                # ajout à gauche
deque([5, 0, 1, 2, 3])
>>> f.popleft()            # suppression à gauche
5
>>> f
deque([0, 1, 2, 3])
>>> len(f)
4
```

3 Parcours de graphes donnés par liste d'adjacence

Les parcours de graphes sont à la base de nombreux algorithmes sur les graphes. Ils vont nous permettre de calculer des plus courts chemins, tester la connexité, tester l'existence d'un cycle (circuit) dans un graphe, etc... On suppose que le graphe est donné par listes d'adjacences, et que l'ensemble de ses sommets est $[\![0, n-1]\!]$.

3.1 Parcours en largeur et plus courts chemins

Une application du **parcours en largeur** est le calcul de plus courts chemins depuis l'origine s_0 du parcours. Il suffit de rajouter une liste dist de distances à la source s_0. Lorsqu'on découvre un nouveau sommet t à partir d'un sommet s, dist[t] prend la valeur dist[s]+1.

```
>>> def parcours_largeur(G,s):
>>> """ graphe non pondéré, distances depuis l'origine s. On utilise une structure de file
>>> fournie par deque. """
>>>     n=len(G)
>>>     inf=float('inf')
>>>     dist=[inf]*n
>>>     F=deque()
>>>     F.append(s)
>>>     while len(F)>0:
>>>         x=F.popleft()
>>>         for y in G[x]:
>>>             if dist[y]==inf:
>>>                 dist[y]=dist[x]+1
>>>                 F.append(y)
>>>     return dist
```

Dans le code précédent, on a utilisé le flottant infini comme initialisation. Tous les sommets à distance finie de s_0 auront après le parcours une valeur associée dans dist qui est positive : c'est la distance d'un plus court chemin entre s_0 et ce sommet.

3.2 Parcours en profondeur

Le **parcours en profondeur** d'un graphe consiste à s'enfoncer le plus possible dans le graphe avant de remonter vers des sommets déjà vus, dont les voisins n'ont pas tous été découverts. Deux variantes du parcours en profondeur sont au programme et permettent de résoudre les problèmes suivants :

- Test de la connexité d'un graphe non orienté ;
- Détection de cycle dans un graphe non orienté.

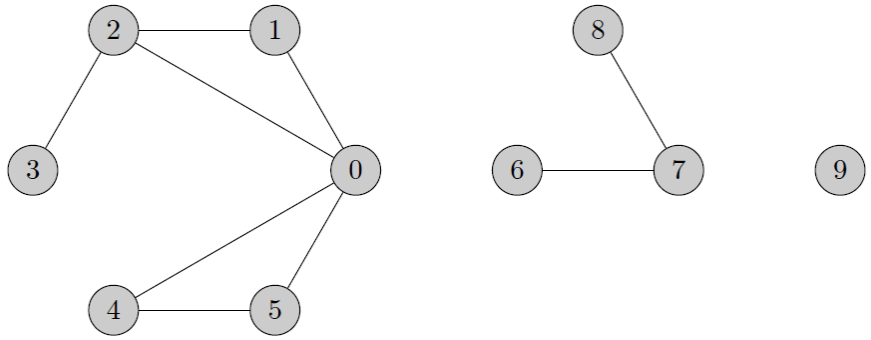
Connexité d'un graphe non orienté

Un parcours en profondeur élémentaire lancé sur un sommet permet de découvrir tous les sommets accessibles depuis ce sommet. Pour tester la connexité, il suffit donc d'effectuer un parcours élémentaire et de vérifier que l'on a atteint les n sommets.

Plus généralement, chaque parcours élémentaire lancé dans la boucle for permet de découvrir un ensemble de sommets accessibles les uns les autres (et aucun autre sommet n'est accessible) : un tel ensemble de sommets s'appelle une **composante connexe**. L'algorithme suivant calcule les composantes connexes d'un graphe non orienté. Il complète simplement le parcours « théorique » avec une liste de listes.

```
>>> def compo_connexes(G):
>>> """ Parcours en profondeur pour calculer les composantes connexes """
>>>     n=len(G)
>>>     deja_vus = [False]*n
>>>     liste_cc = []
>>>     def pp(v):
>>>             deja_vus[v]=True
>>>             for w in G[v]:
>>>                     if not deja_vus[w]:
>>>                             pp(w)
>>>             liste_cc[-1].append(v)
>>>     for i in range(n):
>>>             if not deja_vus[i]:
>>>                     liste_cc.append([])
>>>                     pp(i)
>>>     return liste_cc
```

Par exemple, voici un graphe et les résultats de l'application de cet algorithme.




```
>>> G = [[1, 2, 4, 5], [0, 2], [0, 3], [2], [0, 5], [0, 4], [7], [6, 8], [7], []]
>>> compo_connexes(G)
[[3, 2, 1, 5, 4, 0], [8, 7, 6], [9]]
```

Détection de cycle dans un graphe non orienté

Du point de vue de la connexité dans un graphe non orienté, une arête d'un cycle peut être supprimée sans nuire à la connexité. Dans une application où les arêtes coûtent des ressources (une route, un câble électrique, etc...), et qu'on ne s'intéresse qu'à la connexité, il est utile de savoir repérer l'existence d'un circuit, cela montre que l'on peut économiser des ressources.

Pour détecter l'existence d'un circuit, une couleur va être rajoutée aux sommets du graphe pour rendre compte de leur statut durant le parcours :

- un sommet pas encore découvert sera blanc (ou 0) ;
- un sommet v en cours de traitement, c'est-à-dire que l'appel pp(v) est en cours mais ne s'est pas terminé, sera gris (ou 1) ;
- un sommet v dont le traitement est terminé, c'est-à-dire que l'appel pp(v) a été effectué et s'est terminé, sera noir (ou 2).

La seule difficulté va résider dans le fait que dans un graphe non orienté, une arête $\{u, v\}$ est encodée à la manière de deux arcs $u \to v$ et $v \to u$: le circuit $u \to v \to u$ ne doit pas être compté. Pour ce faire, il faut indiquer que le sommet u a permis la découverte de v. Un graphe possède alors un cycle si et seulement si, lors du parcours en profondeur, un des appels pp(v,parent) trouve un sommet gris dans la liste d'adjacence de v. Voici le code :

```
>>> def sans_cycle(G):
>>>     n=len(G)
>>>     couleurs = [0] * n          #blanc, gris, noir = 0, 1, 2
>>>     def pp(v, parent):
>>>             couleurs[v]=1
>>>             for w in G[v]:
>>>                     if couleurs[w]==0:
>>>                             if not pp(w,v):
>>>                                     return False
>>>                     elif couleurs[w]== 1 and parent!=w:      #on ignore l'arc v -> parent
>>>                             return False
>>>             couleurs[v]=2
>>>             return True
>>>     for i in range(n):
>>>             if couleurs[i] == 0:
>>>                     if not pp(i, None):         #le paramètre parent initial n'est pas pertinent
>>>                             return False
>>>     return True
```

4 Graphes pondérés

Dans le cas des graphes pondérés, les arcs / arêtes sont munis d'un **poids**. Les applications sont nombreuses, car un poids sur un arc peut symboliser une capacité (quantité maximale que l'on peut faire transiter sur l'arc dans une unité de temps), ou un coût (durée ou distance d'un trajet, quantité de carburant nécessaire, etc...).

Toutes le notions vues précédemment sont applicables en prenant soin d'intégrer la notion de poids pour les arcs / arêtes. D'ailleurs, tous les arcs / arêtes peuvent être considérés comme ayant un poids de valeur unitaire dans le cas de graphes non pondérés.

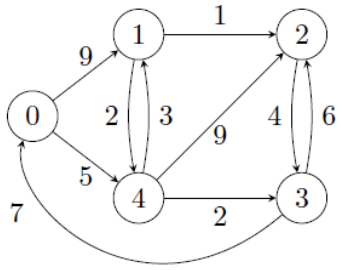
Poids d'un chemin : Pour $p = v_0, v_1, \dots, v_n$ un chemin dans un graphe, on définit son poids comme :

$$\omega(p) = \sum_{i=0}^{n-1} \omega(v_i, v_{i+1})$$

Dans le cas d'une implémentation creuse, il suffit dans la liste d'adjacence d'un sommet u, de stocker des couples (v, p) à la place du seul sommet v : ceci signifie que $(u, c) \epsilon E$ et $\omega(u, v) = p$.

Dans le cas d'une implémentation dense, on utilise la généralisation de la fonction ω à tout couple de sommets. La matrice d'adjacence devient une matrices à valeurs dans $\mathbb{R} \cup \{+\infty\}$.

La figure suivante présente un exemple de graphe pondéré orienté :



En Python, sa représentation par liste d'adjacence serait par exemple :

```
G = [[(1, 9), (4, 5)], [(2, 1), (4, 2)], [(3, 4)], [(0, 7), (2, 6)], [(1, 3), (2, 9), (3, 2)]]
```

Alors que sa matrice d'adjacence et sa représentation en Python (sous forme de liste de listes) est la suivante, avec $inf = float('inf')$:

$$\begin{pmatrix} 0 & 9 & +\infty & +\infty & 5 \\ +\infty & 0 & 1 & +\infty & 2 \\ +\infty & +\infty & 0 & 4 & +\infty \\ 7 & +\infty & 6 & 0 & +\infty \\ +\infty & 3 & 9 & 2 & 0 \end{pmatrix}$$

```
G = [[0, 9, inf, inf, 5],
     [inf, 0, 1, inf, 2],
     [inf, inf, 0, 4, inf],
     [7, inf, 6, 0, inf],
     [inf, 3, 9, 2, 0]]
```

5 Algorithmes de Dijkstra et A*

5.1 Algorithme de Dijkstra

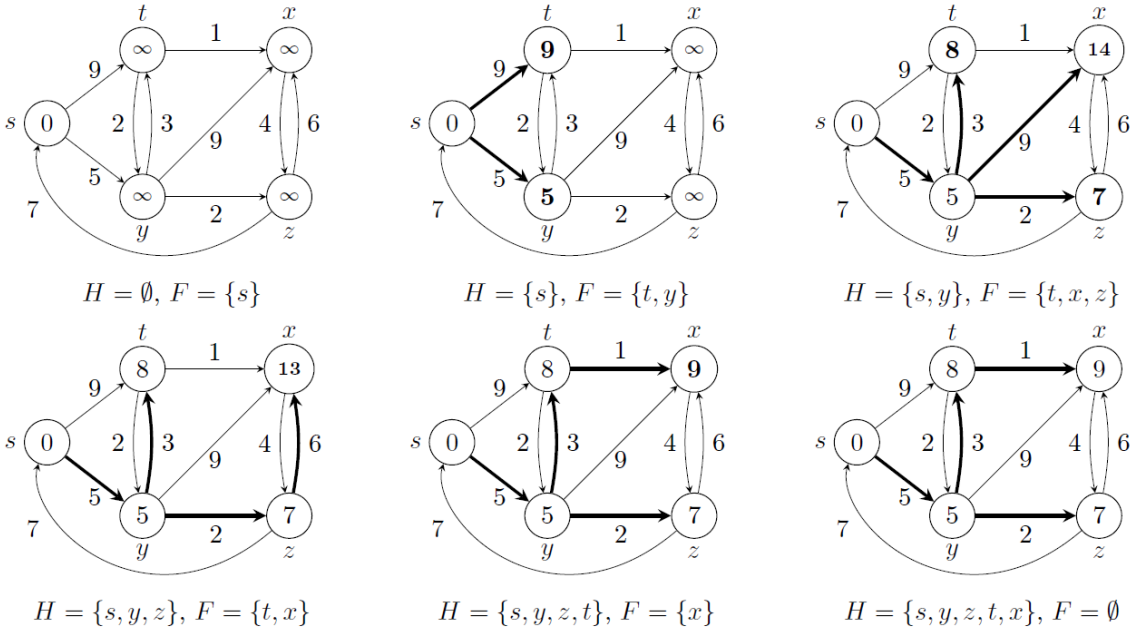
L'algorithme de Dijkstra est une généralisation du parcours en largeur : il consiste donc également à traiter les sommets un par un, par poids depuis l'origine s. C'est donc un **algorithme glouton**.

En supposant que le graphe est représenté par matrice d'adjacence, une implémentation possible est fournie ci-après. On utilise une fonction annexe permettant de trouver le sommet i vérifiant $d[i]$ minimal parmi les sommets tels que $traites[i]$ est faux (s'il existe un tel sommet vérifiant $d[i] < +\infty$, sinon -1 est renvoyé).

```
>>> def cherche_min(d, traites):
>>> """ Renvoie le sommet i vérifiant d[i] minimal et traites[i] faux, s'il existe un tel sommet
>>> tel que d[i] != inf. Sinon, renvoie -1 """
>>>     n=len(d)
>>>     x=-1
>>>     for i in range(n):
>>>         if not traites[i] and d[i] != float('inf') and (x==-1 or d[x]>d[i]):
>>>             x=i
>>>     return x
```

```
>>> def dijkstra_mat(G,s):
>>> """ G donné par matrice d'adjacence. Renvoie les poids chemins de plus petits poids depuis s. """
>>>     n=len(G)
>>>     d = [float('inf')]*n
>>>     d[s]=0
>>>     traites = [False]*n
>>>     while True:
>>>             x=cherche_min(d,traites)
>>>             if x==-1:
>>>                     return d
>>>             for i in range(n):
>>>                     d[i]=min(d[i], d[x]+G[x][i])
>>>             traites[x]=True
```

Exemple du déroulement de l'algorithme de Dijkstra :



Si on utilise un tableau de booléens pour marquer les éléments de F, retirer l'élément de F vérifiant $d[u]$ minimal a un coût $O(n)$. Cette action est effectuée au plus n fois pour un coût total $O(n^2)$. Ce coût majore le reste de l'algorithme, que le graphe soit implémenté via des listes d'adjacences ou une matrice d'adjacences. Ainsi l'algorithme de Dijkstra a un coût $\boldsymbol{O(n^2)}$.

5.2 Algorithme A*

Dans la pratique (par exemple, une recherche d'itinéraire via un outil comme Google Maps ou Waze), tous les poids depuis la source ne nous intéressent pas, on veut surtout le poids minimal entre deux sommets du graphe s et t (et un chemin de plus petit poids associé).

Pour optimiser l'algorithme de Dijkstra, on peut s'arrêter dès que le sommet t est traité dans la boucle principale du parcours, car on sait que d[t] n'évoluera plus et donc éviter des calculs inutiles.

Cette optimisation permet un gain de temps certain la plupart du temps, mais pas dans le pire cas (par exemple si t est le dernier sommet traité !).

Par exemple, dans une recherche d'itinéraire le plus rapide entre Nice et Paris, on se doute qu'il est plutôt inutile d'effectuer des calculs vers Rome ou Barcelone (alors qu'ils sont plus proches en temps de trajet, donc seront examinés par l'algorithme de Dijkstra !).

L'algorithme A* essaie d'éviter des calculs inutiles via une **heuristique** (méthode de résolution de problèmes qui ne repose pas sur l'examen détaillé du dit problème) qui dans notre exemple privilégierait plutôt la direction nord-ouest et éviterait de considérer Rome ou Barcelone.

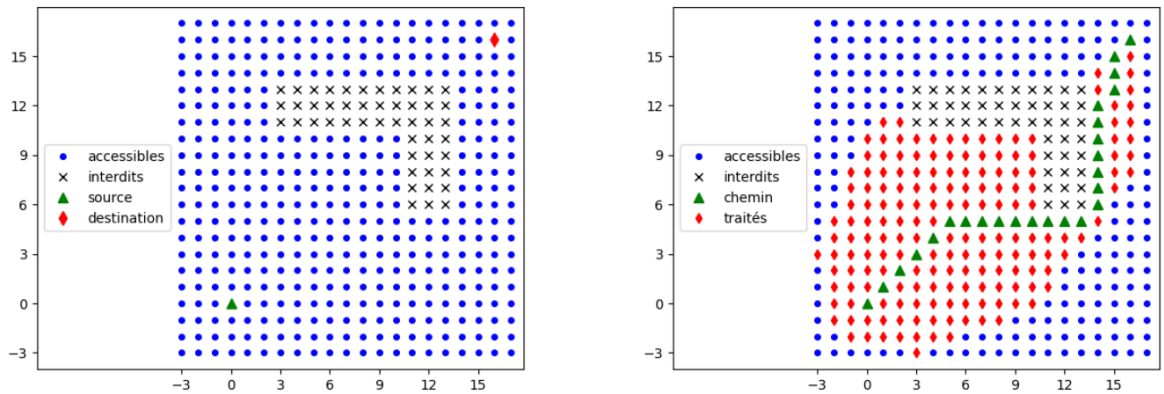
L'idée de l'algorithme A* est la suivante : on suppose connue une fonction f positive (l'heuristique), qui donne pour un nœud u une idée du poids du plus court chemin de u à t. Une telle heuristique est dite admissible si elle ne surestime jamais cette distance, c'est-à-dire que pour tout sommet u du graphe, on ait : $0 \leq f(u) \leq \delta(u;\ t)$.

L'algorithme fonctionne alors comme celui de Dijkstra, à l'exception du fait que l'on retire de F un sommet v tel que $f(v) + d[v]$ est minimal (si f est la fonction identiquement nulle, on retrouve l'algorithme de Dijkstra) dans la boucle principale. Voici deux exemples :

- Sur un graphe constitué de points du plan dont certains sont reliés par des segments, où l'on cherche à relier deux sommets s et t par un enchaînement le plus court possible, une fonction f possible serait la distance euclidienne entre le point considéré et la destination t : par inégalité triangulaire, f vérifie bien l'hypothèse $f(u) \leq \delta(u;\ t)$ pour tout sommet u du graphe et est donc admissible.
- Si on cherche à minimiser le temps de parcours en voiture, en supposant que l'on roule toujours à la vitesse maximale autorisée sur chacune des routes, une heuristique f possible serait la distance à vol d'oiseau divisée par la vitesse maximale sur autoroute (130 km/h).

Un exemple d'utilisation de A* (provenant de la page Wikipedia dédiée) :

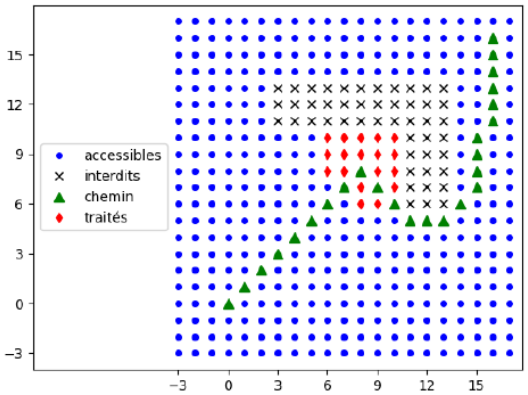
Considérons la grille infinie $\mathbb{Z}^2$, dans laquelle on peut se déplacer en une étape d'un sommet vers un de ces huit voisins (un cran horizontalement ou verticalement, ou en diagonale). Le coût d'un déplacement est celui de la distance parcourue (1 ou $\sqrt{2}$). Il faut trouver un plus court chemin du point $(0; 0)$ au point $(16; 16)$, sachant qu'une zone constituée de deux rectangles est interdite au déplacement. La figure suivante présente la grille du problème et le résultat fourni par A*, respectivement à gauche et à droite.



la fonction f utilisée associe simplement à un nœud sa distance euclidienne à la destination $(16;\ 16)$. Les triangles donnent le chemin trouvé par l'algorithme, de longueur environ 27,31. Il y a 196 sommets en comptant ceux du chemin qui ont été traités. A ceux-ci s'ajoutent 95 sommets découverts mais qui auraient été traités après que la destination $(16;\ 16)$ le soit.

L'algorithme de Dijkstra (ou l'algorithme A* avec une fonction f identiquement nulle), traite 2027 sommets (et en découvre 231 supplémentaires), soit environ 10 fois plus de sommets. De plus, le chemin obtenu à l'aide de l'algorithme de Dijkstra a la même longueur que celui trouvé par l'algorithme A*.

Si on se donne une fonction f non admissible, on trouvera en général un chemin encore plus rapidement, mais non optimal. Voici les sommets traités et le chemin trouvé par l'algorithme A* dans l'exemple précédent lorsque la fonction f vaut 5 fois la distance euclidienne entre le nœud et la destination :



Ce chemin est de longueur environ 29,8, un peu plus que le chemin optimal. Mais l'algorithme traite beaucoup moins de sommets !

Table des matières