Chapitre ITC5

Graphes

Plan

1. Introduction et vocabulaire
   - Quelques problèmes classiques
   - Graphes non orientés
   - Graphes orientés
   - Étiquetages et pondérations
2. Chemins, connexités, et parcours
   - Chemins
   - Connexité
   - Parcours
3. Recherche de plus court chemin

Section 1

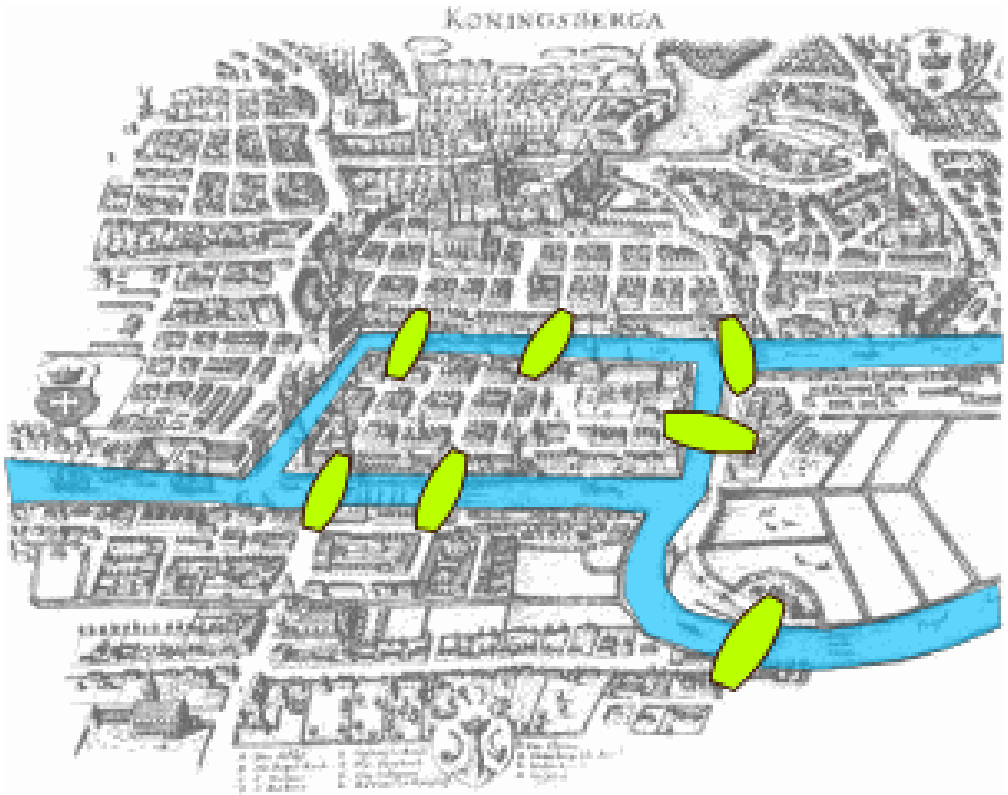
Introduction et vocabulaire

1 Introduction et vocabulaire
- Quelques problèmes classiques
- Graphes non orientés
- Graphes orientés
- Étiquetages et pondérations

2 Chemins, connexités, et parcours

3 Recherche de plus court chemin

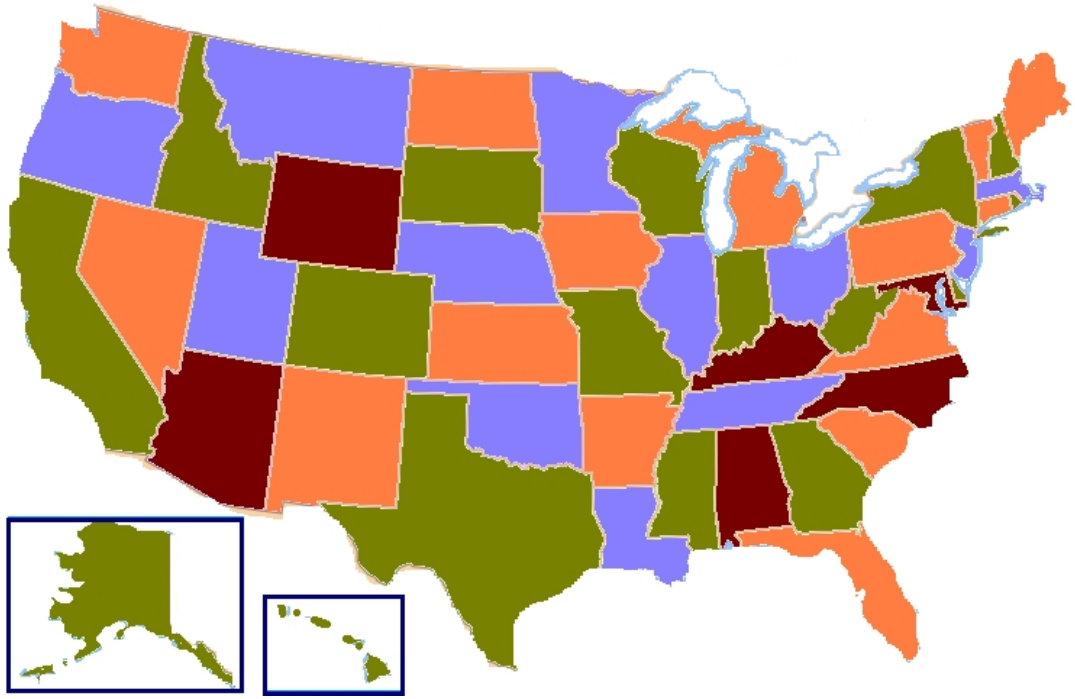
Les ponts de Konigsberg



(Euler, 1736)

Puis-je me promener dans cette ville en passant une et une seule fois sur chaque pont ?

Le théorème des quatre couleurs



(Appel & Haken, 1976)

Peut-on colorer une carte avec quatre couleurs seulement ?

- Première preuve assistée par ordinateur.

Le voyageur de commerce

Problème ouvert

Trouver efficacement un cycle minimal passant par toutes les villes.

Les graphes

Ces problèmes se résolvent à l'aide de la **théorie des graphes**, un domaine central de l'informatique

Les utilisations des graphes sont innombrables :

- GPS, Cartes : recherches d'itinéraires
- Conception de plannings complexes
- Réseaux : internet, réseaux sociaux, propagation d'épidémies, énergie
- Optimisation du transport de marchandises
- Biologie : structures moléculaires, organisation du vivant
- ...

Une algorithmique efficace est nécessaire

Graphe	**Nœuds**	**Liens**
Kœnigsberg	4	7
OpenStreetMap [1]	$6,6 \times 10^9$	730×10^6
Internet (www) [2]	5×10^9	750×10^9
Facebook [3]	2×10^9	750×10^9

1. Statistiques en ligne 2020
2. *www.worldwidewebsize.com* from Tilburg University ; avec en moyenne 70 liens pas page
3. En moyenne, 338 amis/personne

Une algorithmique efficace est nécessaire

Graphe	Nœuds	Liens
Kœnigsberg	4	7
OpenStreetMap [1]	$6,6 \times 10^9$	730×10^6
Internet (www) [2]	5×10^9	750×10^9
Facebook [3]	2×10^9	750×10^9

Besoin d'algorithmes efficaces

Un algorithme linéaire traitant chaque profil Facebook en 1s prendrait plus de 20 ans...

1. Statistiques en ligne 2020
2. *www.worldwidewebsize.com* from Tilburg University ; avec en moyenne 70 liens pas page
3. En moyenne, 338 amis/personne

1 Introduction et vocabulaire
- Quelques problèmes classiques
- Graphes non orientés
- Graphes orientés
- Étiquetages et pondérations

2 Chemins, connexités, et parcours

3 Recherche de plus court chemin

Graphe non orienté

Définition

Un **graphe non-orienté** est un couple $G = (S, A)$ où

- S est un ensemble (fini) appelé ensemble des **sommets** ou des **nœuds**,
- A est un ensemble de paires de sommets, appelées **arêtes**.

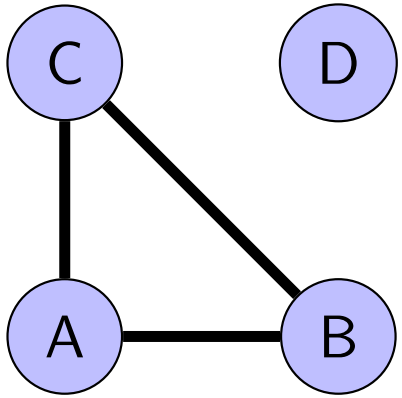
- Une arête a donc pour forme $\{x, y\}$ avec $x, y \in S$.
- L'ordre n'a pas d'importance car $\{x, y\} = \{y, x\}$.
- **Notation :** on notera simplement $xy \in A$ le fait que l'**arête** $\{x, y\} \in A$.

Notation pour les complexités

On s'efforcera de noter $n = |S|$ et $m = |A|$.

Représentation graphique

Un **graphe non orienté** se représente comme ceci :



- Les nœuds sont représentés par des boîtes généralement circulaires.
- Une arête est représentée par une ligne reliant deux sommets.

Ici le graphe est $G = (S, A)$ avec

- $S = \{A, B, C, D\}$
- $A = \{\{A, B\}, \{A, C\}, \{B, C\}\}$

l'emplacement des nœuds sur la représentation graphique n'a pas d'importance

Représentation : matrice d'adjacence

Comment représenter informatiquement un graphe ?

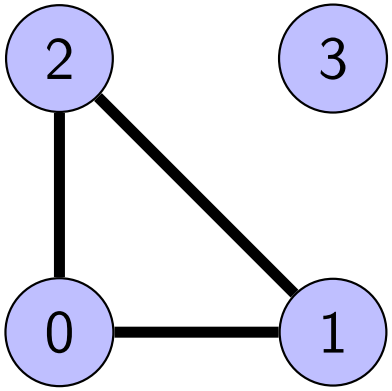
Représentation par matrice d'adjacence

Un graphe G peut se représenter par une matrice M de taille $n \times n$ telle que $ij \in A \Leftrightarrow M_{i,j} = 1$.

En Python, on prendra par exemple un tableau 2D Numpy avec des 0 et des 1.

- **Remarque** : M est symétrique
- **Avantages**
  - Vérification de lien en temps $O(1)$
  - Ajout/suppression d'une arête facile en temps $O(1)$
- **Inconvénients**
  - Place en mémoire $O(n^2)$
  - Énumération des voisins en temps $O(n)$

Représentation par matrice d'adjacence : exemple



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

```
1  s=[0,1,2,3] # définition des noeuds
2  m=numpy.zeros([4,4]) # matrice d'adjacence
3  m[0,1]=1
4  m[1,0]=1
5  m[0,2]=1
6  m[2,0]=1
7  m[1,2]=1
8  m[2,1]=1
```

Modification du graphe par matrice d'adjacence

- Lecture d'arête

```
1 def a_pour_voisin(m,i,j):
2     return (m[i,j]==1)
```

- Ajout d'arête

```
1 def ajoute_arete(m,i,j):
2     m[i,j]=1
3     m[j,i]=1
```

- Suppression d'arête

```
1 def enleve_arete(m,i,j):
2     m[i,j]=0
3     m[j,i]=0
```

- Liste des voisins :

```
1 def liste_voisins(m,i):
2     voisins=[]
3     for j in range(m.shape[1]):
4         if m[i,j]==1:
5             voisins.append(j)
6     return voisins
```

Représentation par listes d'adjacence

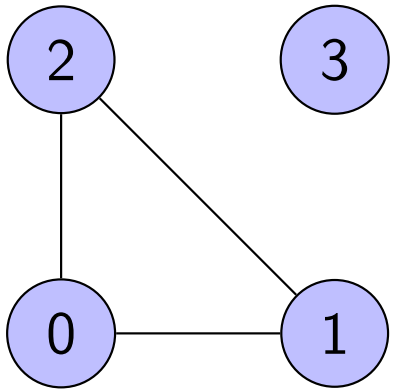
Comment résoudre le problème d'espace occupé par la matrice d'adjacence ?

Représentation par listes d'adjacences

Un graphe G peut se représenter par un tableau T de taille n où chaque case i contient la liste des voisins de i.

- **Remarque** : c'est mieux si les listes sont triées.
- **Avantages**
  - Place en mémoire $O(n+m)$
  - Liste des voisins $O(1)$
- **Inconvénients**
  - Vérification de lien en temps $O(n)$
  - Ajout/suppression d'une arête en temps $O(n)$

Représentation par listes d'adjacence : exemple



$$T = \begin{bmatrix} [1,2] & [0,2] & [0,1] & [] \end{bmatrix}$$

```
1 s=[0,1,2,3] # liste des noeuds
2 t=[[1,2],[0,2],[0,1],[]] # liste des voisins
```

Remarque : Gain d'espace : les "0" inutiles ont disparu.

Modification du graphe par listes d'adjacence

- Lecture d'arête

```
1 def a_pour_voisin(t,i,j):
2     voisins=t[i]
3     for k in range(len(voisins)):
4         if voisins[k]==j:
5             return True
6     return False
```

- Ajout d'arête

```
1 def ajoute_arete(t,i,j):
2     if not(a_pour_voisin(t,i,j)):
3         t[i].append(j)
4         t[j].append(i)
```

- Suppression d'arête

```
1 def enleve_arete(t,i,j):
2     if a_pour_voisin(t,i,j):
3         t[i].remove(j)
4         t[j].remove(i)
```

- Liste des voisins :

```
1 def liste_voisins(a,i):
2     return t[i]
```

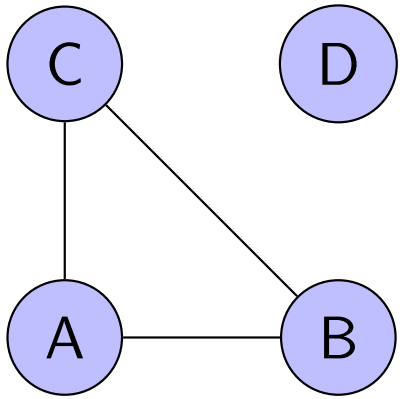
Voisinage et degré d'un sommet

Voisinage

Les voisins d'un sommet $x \in S$ est l'ensemble $\mathcal{V}(x) = \{y \in S | xy \in A\}$.

Degré

Le *degré* d'un sommet $x \in S$ est défini par $d(x) = |\mathcal{V}(x)|$.



Le degré de A, B et C est 2, celui de D est 0.

Un nœud de degré 0 est dit **isolé**.

Bilan : que choisir ?

Matrice d'adjacence ou listes d'adjacence ?

Bilan : que choisir ?

Matrice d'adjacence ou listes d'adjacence ?

	Matrice d'adjacence	**Listes d'adjacence**
Espace	$O(n^2)$	$O(n+m)$
$xy \in A$?	$O(1)$	$O(n)$
ajout/suppression sommet	$O(n^2)$	$O(n+m)$
ajout/suppression arête	$O(1)$	$O(n)$
liste des voisins	$O(n)$	$O(1)$

Bilan : que choisir ?

Matrice d'adjacence ou listes d'adjacence ?

	Matrice d'adjacence	Listes d'adjacence
Espace	$O(n^2)$	$O(n+m)$
$xy \in A$?	$O(1)$	$O(n)$
ajout/suppression sommet	$O(n^2)$	$O(n+m)$
ajout/suppression arête	$O(1)$	$O(n)$
liste des voisins	$O(n)$	$O(1)$

- **Cela dépend de la densité du graphe :**
  - graphe dense ($m \simeq n^2$) : le gain d'espace par listes d'adjacence est négligeable $\rightarrow$ **Matrice d'adjacence**
  - graphe creux ($m << n^2$) $\rightarrow$ **Listes d'adjacence**
  - pour les grands graphes (Facebook, réseau routier, etc) : $n > 10^6 \Rightarrow n^2 > 10^{12} \simeq$ Tera : on peut difficilement s'en sortir avec une matrice.

Bilan : que choisir ?

- **Cela dépend aussi des algorithmes utilisés :** si on a besoin de listes de voisins (cas des parcours) on préfère les listes d'adjacence.

Important

Les complexités spatiales ou temporelles des algorithmes sur les graphes **dépendent** de la représentation choisie.

- Soit le sujet précise la représentation à utiliser et il faut en tenir compte.
- Soit vous devez préciser avant tout la représentation choisie.

1 Introduction et vocabulaire
- Quelques problèmes classiques
- Graphes non orientés
- Graphes orientés
- Étiquetages et pondérations

2 Chemins, connexités, et parcours

3 Recherche de plus court chemin

Graphes orientés

- Pourquoi ?
  - Voie à sens unique
  - Marchandises ne pouvant circuler que dans un sens
  - Page internet pointant vers une autre
  - Relation d'amitié sur les réseaux sociaux $\rightarrow$ non symétrique ☺ ♡ ☹

Graphes orientés

- Pourquoi ?
  - Voie à sens unique
  - Marchandises ne pouvant circuler que dans un sens
  - Page internet pointant vers une autre
  - Relation d'amitié sur les réseaux sociaux $\rightarrow$ non symétrique ☺ ♡ ☹

Définition

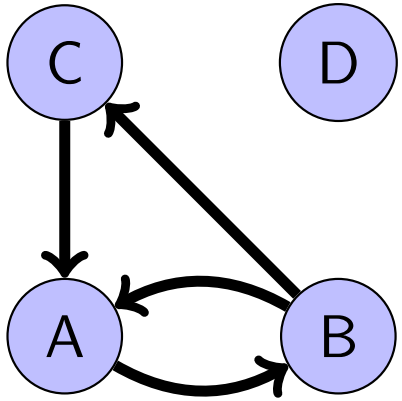
Un **graphe orienté** (ou digraphe) est un couple $G = (S, A)$ où

- S est un ensemble (fini) appelé ensemble des **sommets** ou des **nœuds**,
- A est un ensemble de **couples** de sommets, appelés **arcs**.

- Un arc a donc pour forme (x, y) avec $x, y \in S$.
- L'ordre est important car $(x, y) \neq (y, x)$.
- **Notation :** on notera aussi simplement $xy \in A$ le fait que l'**arc** $(x, y) \in A$.

Représentation graphique

Un graphe orienté se représente comme ceci :



- Les nœuds sont représentés par des boîtes généralement circulaires.
- Une arc (x, y) est représentée par une flèche reliant x à y.

Ici le graphe est $G = (S, A)$ avec

- $S = \{A, B, C, D\}$
- $A = \{(A, B), (B, A), (C, A), \{B, C\}\}$

Voisinage orienté

Dans un graphe orienté on a les voisins entrants et sortants.

Voisinage

- Les voisins sortants d'un sommet $x \in S$ est l'ensemble $\mathcal{V}^+(x) = \{y \in S | xy \in A\}$.
- Les voisins entrants d'un sommet $x \in S$ est l'ensemble $\mathcal{V}^-(x) = \{y \in S | yx \in A\}$.
- Les voisins d'un sommet $x \in S$ est l'ensemble $\mathcal{V}(x) = \mathcal{V}^+(x) \cup \mathcal{V}^-(x)$.

Voisinage orienté

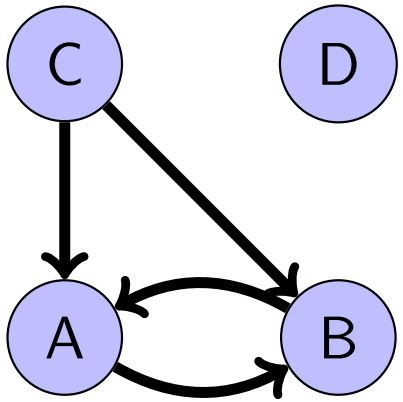
On définit ainsi le degré entrant et le degré sortant de chaque sommet

Degré

- Le degré sortant d'un sommet $x \in S$ est l'ensemble $d^+(x) = |\mathcal{V}^+(x)|$.
- Les degré entrant d'un sommet $x \in S$ est l'ensemble $d^-(x) = |\mathcal{V}^-(x)|$.

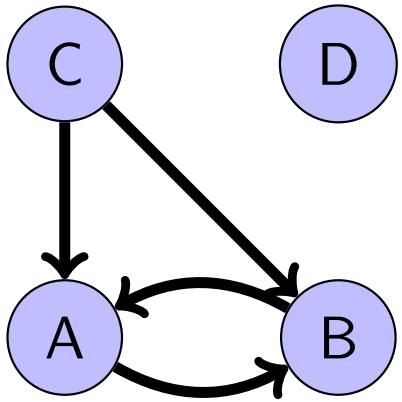
Voisinage orienté : exemple

Donner les voisinages et degrés entrants et sortants dans ce graphe



Voisinage orienté : exemple

Donner les voisinages et degrés entrants et sortants dans ce graphe



On remarque la propriété suivante :

Proposition

Dans tout graphe orienté

$$\sum_{x \in S} d^+(x) = \sum_{x \in S} d^-(x) = |A|.$$

Représentation : matrice d'adjacence

Comment représenter les graphe orientés ?

Représentation : matrice d'adjacence

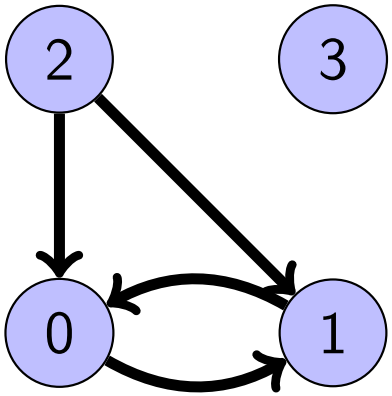
Comment représenter les graphe orientés ?

Représentation par matrice d'adjacence

Un graphe orienté G peut se représenter par une matrice booléenne M de taille $n \times n$ telle que $ij \in A \Leftrightarrow M_{i,j} = Vrai$.

- **Remarque** : M n'est plus nécessairement symétrique.
- **Avantages**
  - Vérification de lien en temps $O(1)$
  - Ajout/suppression d'une arc facile en temps $O(1)$
- **Inconvénients**
  - Place en mémoire $O(n^2)$
  - Énumération des voisins entrants et sortants en temps $O(n)$

Représentation par matrice d'adjacence : exemple



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

```
1 s=[0,1,2,3]
2 m=numpy.zeros([4,4])
3 m[0,1]=1
4 m[1,0]=1
5 m[2,0]=1
6 m[2,1]=1
```

Modification du graphe par matrice d'adjacence

- Ajout d'arête

```
1 def ajoute_arete(m,i,j):
2     m[i,j]=1
```

- Suppression d'arête

```
1 def enleve_arete(m,i,j):
2     m[i,j]=0
```

- Liste des voisins entrants :

```
1 def liste_voisins_entrants(m,i):
2     voisins=[]
3     for j in range(m.shape[0]):
4         if m[j,i]==1:
5             voisins.append(j)
6     return voisins
```

- Liste des voisins sortants :

```
1 def liste_voisins_sortants(m,i):
2     voisins=[]
3     for j in range(m.shape[1]):
4         if m[i,j]==1:
5             voisins.append(j)
6     return voisins
```

Représentation par listes d'adjacence

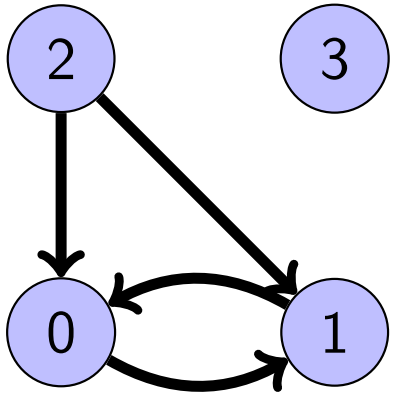
Encore plus de 0 dans la matrice d'adjacence...

Représentation par listes d'adjacences

Un graphe orienté G peut se représenter par un tableau T de taille n où chaque case i contient la liste des voisins **sortants** de i.

- **Remarque** : C'est mieux avec des listes triées.
- **Avantages**
  - Place en mémoire $O(n+m)$
  - Liste des voisins sortants $O(1)$
- **Inconvénients**
  - Vérification de lien en temps $O(n)$
  - Ajout/suppression d'une arc en temps $O(n)$

Représentation par listes d'adjacence : exemple



$$T = \begin{bmatrix} [1] & [0] & [0,1] & [] \end{bmatrix}$$

```
1 s=[0,1,2,3]
2 t=[[1],[],[0,1],[]]
```

Remarque : Gain d'espace : les "0" inutiles ont disparu.

Modification du graphe par listes d'adjacence

- Lecture d'arête

```
1 def a_pour_voisin(t,i,j):
2     voisins=t[i]
3     for k in range(len(voisins)):
4         if voisins[k]==j:
5             return True
6     return False
```

- Ajout d'arête

```
1 def ajoute_arete(t,i,j):
2     if not(a_pour_voisin(t,i,j)):
3         t[i].append(j)
```

- Suppression d'arête

```
1 def enleve_arete(t,i,j):
2     if a_pour_voisin(t,i,j):
3         t[i].remove(j)
```

- Liste des voisins :

```
1 def liste_voisins_sortants(a,i):
2     return t[i]
```

- Liste des voisins entrants : compliqué

Matrice ou listes d'adjacence : que choisir ?

Mêmes remarques

L'orientation des arêtes ne change rien aux remarques précédente sur le choix de la représentation.

Deux remarques quand même :

- Avec l'orientation les graphes ont tendance à être plus creux
- Les algorithmes, donc les complexités, ne sont pas les mêmes que dans le cas non orienté (exemple : lister les voisins)

Lister les voisins

- par matrice d'adjacence

Lister les voisins

- par matrice d'adjacence
  - Voisins sortant de i : parcourir la ligne $i \rightarrow O(n)$
  - Voisins entrants de i : parcourir la colonne $i \rightarrow O(n)$
- par listes d'adjacence

Lister les voisins

- par matrice d'adjacence
  - Voisins sortant de i : parcourir la ligne $i \rightarrow O(n)$
  - Voisins entrants de i : parcourir la colonne $i \rightarrow O(n)$
- par listes d'adjacence
  - Voisins sortants de i : facile $O(1)$
  - Voisins entrants de i : **compliqué** $O(n+m)$
    - Il faut en effet parcourir toutes les listes du tableau pour trouver où apparaît i.

1 Introduction et vocabulaire
- Quelques problèmes classiques
- Graphes non orientés
- Graphes orientés
- Étiquetages et pondérations

2 Chemins, connexités, et parcours

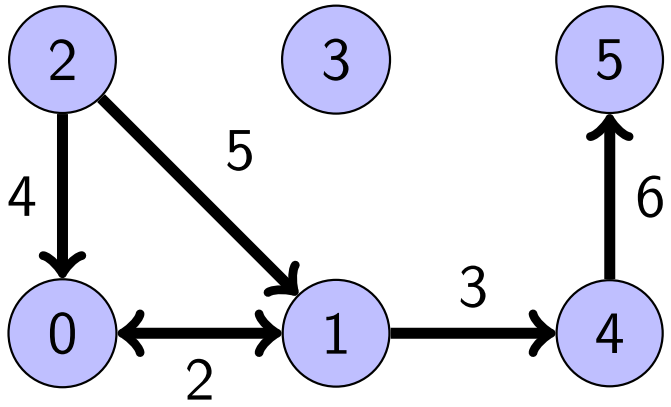
3 Recherche de plus court chemin

Étiquetages et pondérations

On souhaite pouvoir ajouter des informations aux nœuds et/ou aux arêtes/arcs, par exemple des poids ou des coûts.

- Pourquoi ?
  - Distances sur les arcs
  - Coût pour emprunter une voie, établir une voie commerciale, etc.
  - Capacité dans un réseau d'eau/énergie
- Si le graphe est défini par matrice d'adjacence : dans la matrice d'adjacence, on indique les poids des arrêtes/arcs. On ajoute un tableau pour les points des sommets.
- Si le graphe est défini par listes d'adjacence : on crée une autre liste de liste indiquant les poids des arêtes/arcs.

Étiquetages et pondérations : exemple



Matrice d'adjacence

$$M = \begin{pmatrix} 0 & 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 3 & 0 \\ 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Liste d'adjacence

$$T = \begin{bmatrix} [1] & [0,4] & [0,1] & [] & [5] \end{bmatrix}$$

$$p = \begin{bmatrix} [2] & [4,5] & [2,3] & [] & [6] \end{bmatrix}$$

Section 2

Chemins, connexités, et parcours

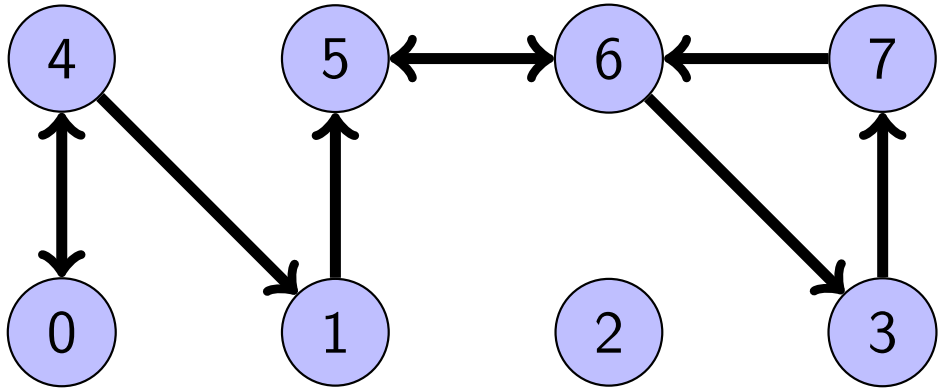
1 Introduction et vocabulaire

2 Chemins, connexités, et parcours

- Chemins
- Connexité
- Parcours

3 Recherche de plus court chemin

Chemins dans un graphe



Chemin dans un graphe (orienté ou non)

Un *chemin* de longueur $N \in \mathbb{N}$ dans un graphe $G = (S, A)$ est une suite de sommets $(x_0, \ldots, x_N) \in S^N$ telle que $\forall i \in [|0, N-1|],\ x_i x_{i+1} \in A$.

- Lorsque $x_0 = x_N$ et $N > 0$ on parle de **cycle**.
- Si les arcs sont pondérés par $p : A \to \mathbb{R}$, le **poids du chemin** est

$$\sum_{i=0}^{N-1} p(x_i, x_{i+1}).$$

Accessibilité

Définition

Soient $x, y \in S$ deux sommets d'un graphe $G = (S, A)$, on note

$$x \rightsquigarrow y$$

et on dit que y est *accessible* depuis x lorsqu'il existe un chemin reliant x à y. La relation $\rightsquigarrow$ est :

1. **réflexive** : $x \rightsquigarrow x$,
2. **transitive** : $x \rightsquigarrow y$ et $y \rightsquigarrow z \Rightarrow x \rightsquigarrow z$.

1 Introduction et vocabulaire

2 Chemins, connexités, et parcours

- Chemins
- Connexité
- Parcours

3 Recherche de plus court chemin

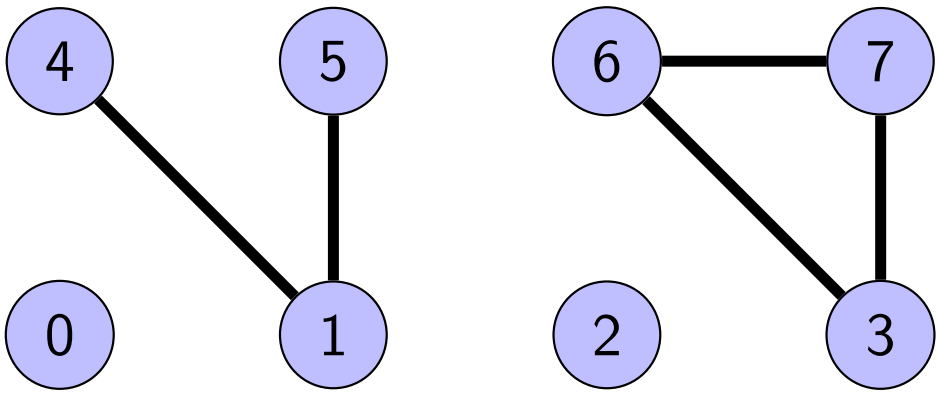
Connexité : cas non orienté

Cas non orienté

- Lorsque G est un graphe non orienté la relation d'accessibilité $\rightsquigarrow$ est symétrique, c'est donc **une relation d'équivalence**.
- On appelle **composantes connexes** les classes d'équivalence de la relation $\rightsquigarrow$.

Ainsi une composante connexe C est un sous-ensemble de sommets $C \subset S$ saturé tel que

$$\forall x, y \in C, x \rightsquigarrow y \text{ et } y \rightsquigarrow x.$$



Connexité : cas orienté

Dans le cas d'un graphe orienté $G = (S, A)$ on distingue deux types de connexité :

- **Composantes connexes** :

Composantes connexes

Dans un graphe orienté, on appelle composante connexe les composantes connexes de G dans lequel on a supprimé l'orientation des arcs.

- **Composantes fortement connexes** :

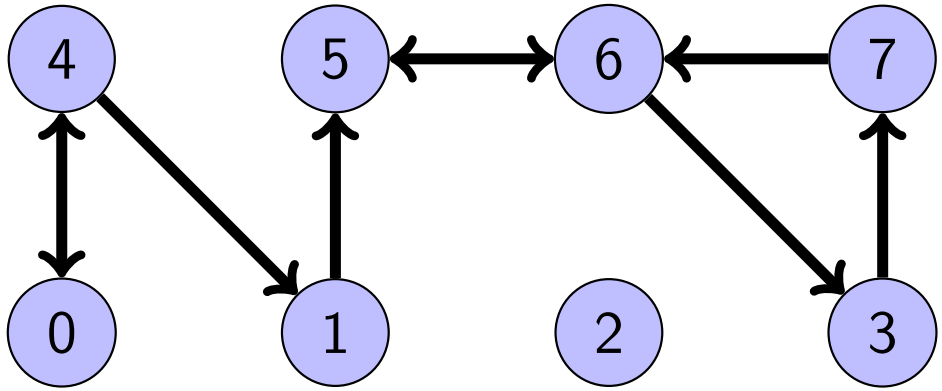
Composantes fortement connexes

Dans un graphe orienté, on appelle **composante fortement connexe** un sous-ensemble de sommets $C \subset S$ saturé tel que

$$\forall x, y \in C, x \rightsquigarrow y \text{ et } y \rightsquigarrow x.$$

Connexité : cas orienté : exemple

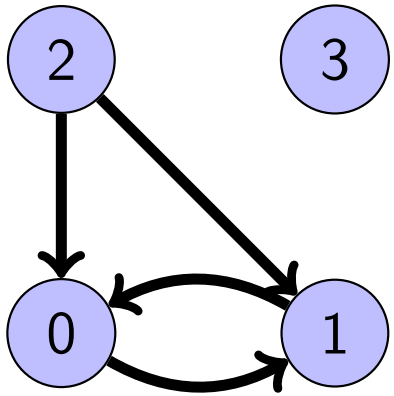
Déterminer les composantes connexes et fortement connexes de ce graphe



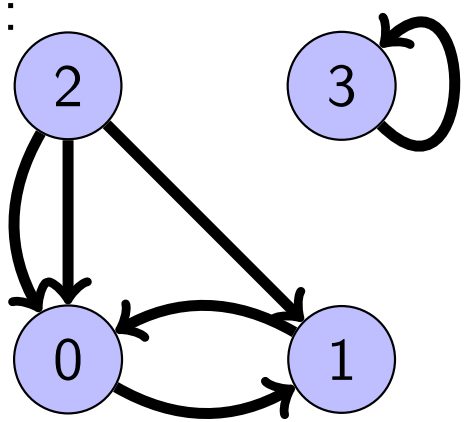
Pour le moment nous n'avons aucun algorithme permettant de déterminer toutes ces connexités.

Quelques graphes particuliers

Graphe simple : graphe ne contenant que des arêtes/arcs simples et pas de boucles (arête/arc d'un sommet sur lui-même) :

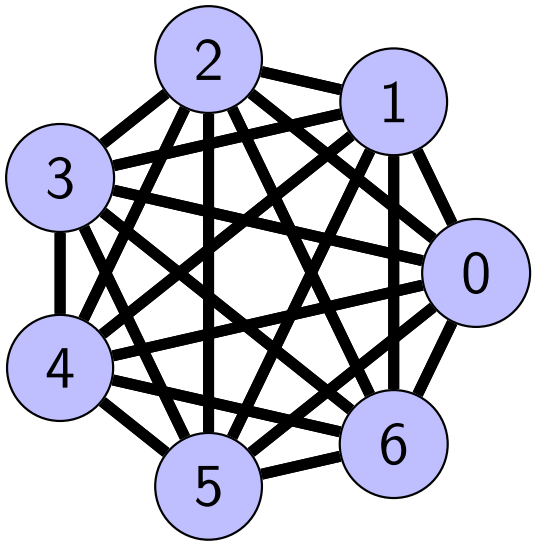


Multigraphe : graphe contenant des boucles ou des arrêtes/arcs multiples :



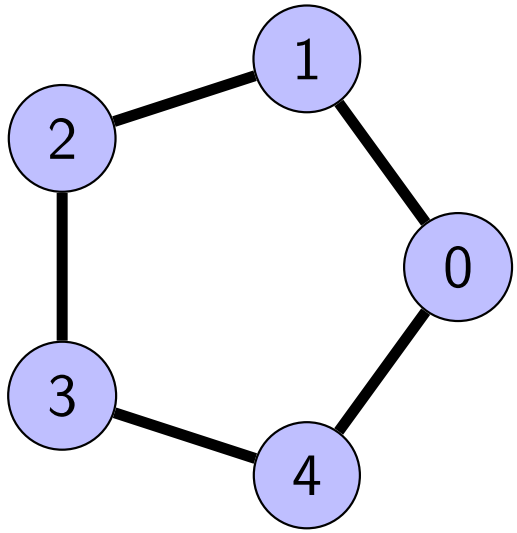
Quelques graphes particuliers

Graphe complet : graphe simple où tous les sommets sont voisins :



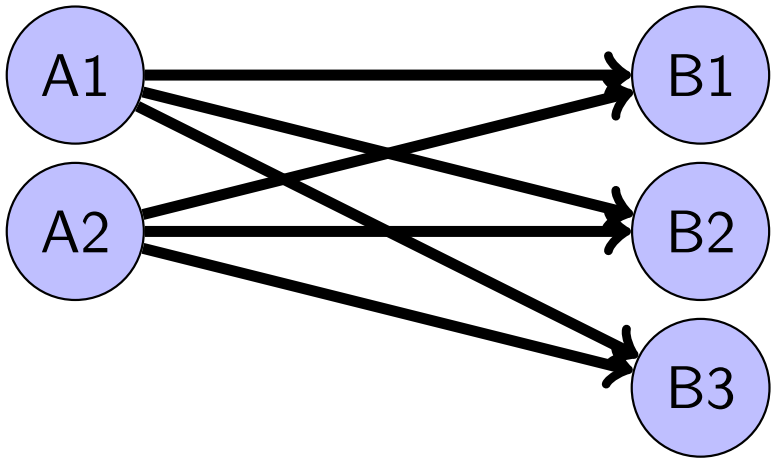
Quelques graphes particuliers

Graphe cycle C_n : graphe connexe, où tous les sommets ont pour degré 2



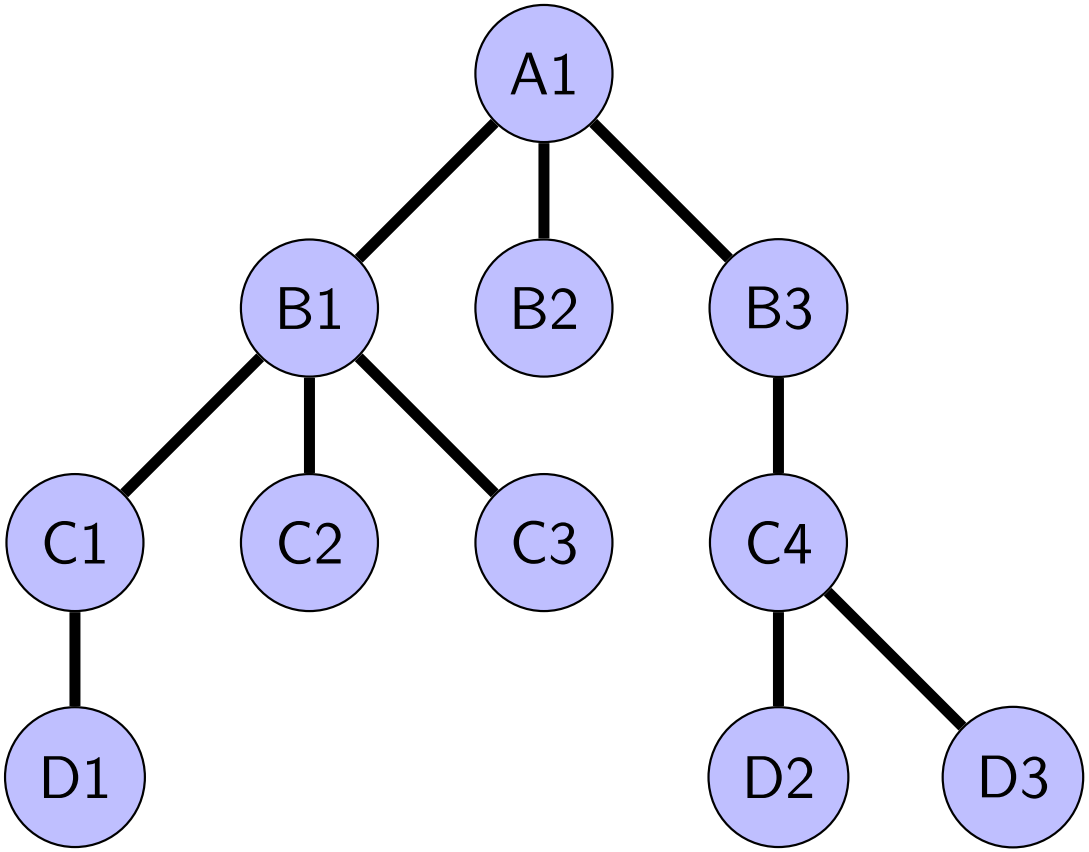
Quelques graphes particuliers

Graphe biparti : $S = A \cup B$ avec un arc (x, y) pour tout $x \in A$ et $y \in B$ et aucun arc entre sommets A et entre sommets B



Quelques graphes particuliers

Arbre : graphe non orienté, acyclique, connexe



Forêt : graphe non orienté, acyclique = plusieurs arbres

1 Introduction et vocabulaire

2 Chemins, connexités, et parcours

- Chemins
- Connexité
- Parcours

3 Recherche de plus court chemin

Parcours dans un graphe

Parcours de graphe

Le parcours d'un graphe consiste à explorer tous les sommets accessibles depuis un sommet x. Le type de parcours détermine dans quel ordre les sommets seront visités.

Nous étudierons deux parcours :

- le **parcours en profondeur** où les sommets à explorer sont placés dans une **pile**
- le **parcours en largeur** où les sommets à explorer sont placés dans une **file**

La **représentation** choisie est celle par **listes d'adjacence**.

Les piles et les files

Une **liste** est une structure de données qui permet de stocker un nombre quelconque de valeurs et d'accéder à n'importe quelle valeur :

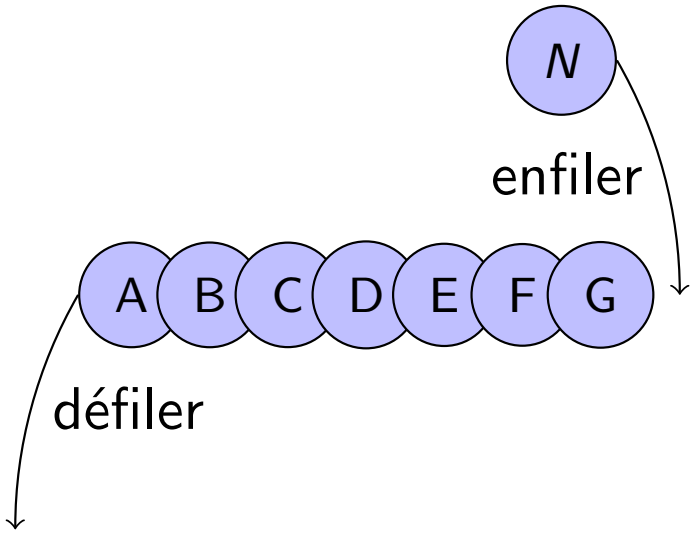
- très souple d'utilisation
- peu performant

Une **pile** ou une **file** est une structure permettant de stocker un nombre quelconque de valeurs, mais la lecture/écriture se fait en un endroit qu'on ne peut pas choisir :

- on peut stocker une valeur sans choisir à quel emplacement
- on peut récupérer une seule valeur (cela dépend du type, pile ou file)
- si c'est bien implémenté, c'est extrêmement performant

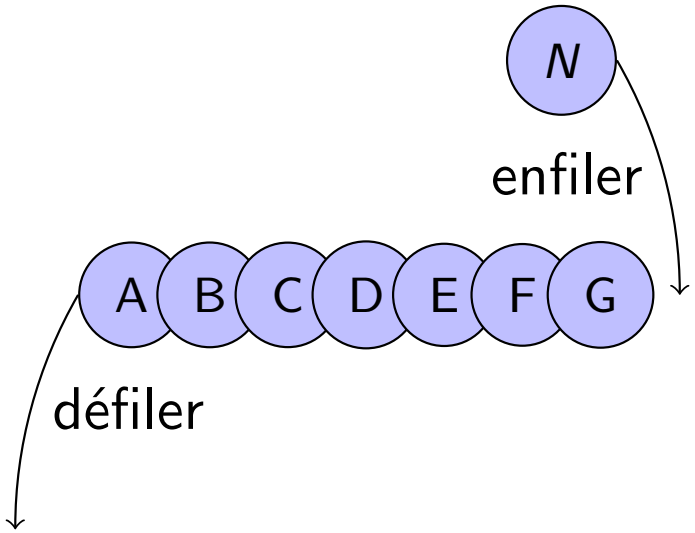
Les files

ou structures **FIFO** : First In First Out



Les files

ou structures **FIFO** : First In First Out

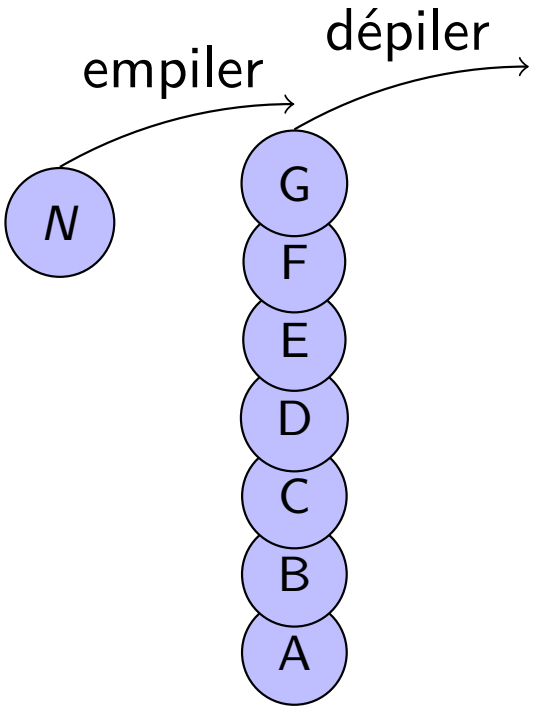


En Python, on peut l'implémenter avec une liste :

- pour enfiler : `file.append(element)`
- pour défiler : `element=file.pop(0)`

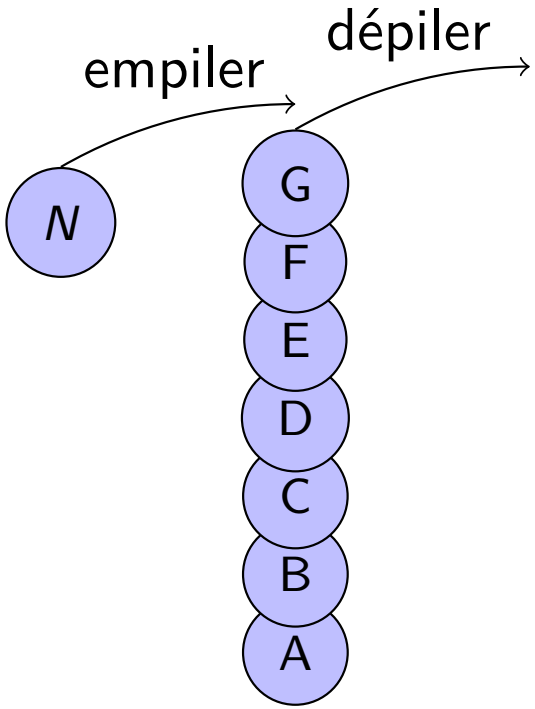
Les piles

ou structures **FILO** : First In Last Out



Les piles

ou structures **FILO** : First In Last Out



En Python, on peut l'implémenter avec une liste :

- pour empiler : `file.append(element)`
- pour dépiler : `element=file.pop()`

Implémentation plus efficace en Python

Si on veut une implémentation efficace, on se tournera vers des librairies spécifiques comme `collections.deque` :

```
1 from collections import deque
2 pile=deque(["A","B","C"])
3 pile.append("D")
4 print(pile.pop())
```

Principe des parcours

Le but est de partir d'un nœud et de découvrir tous les nœuds qu'on peut atteindre depuis ce nœud :

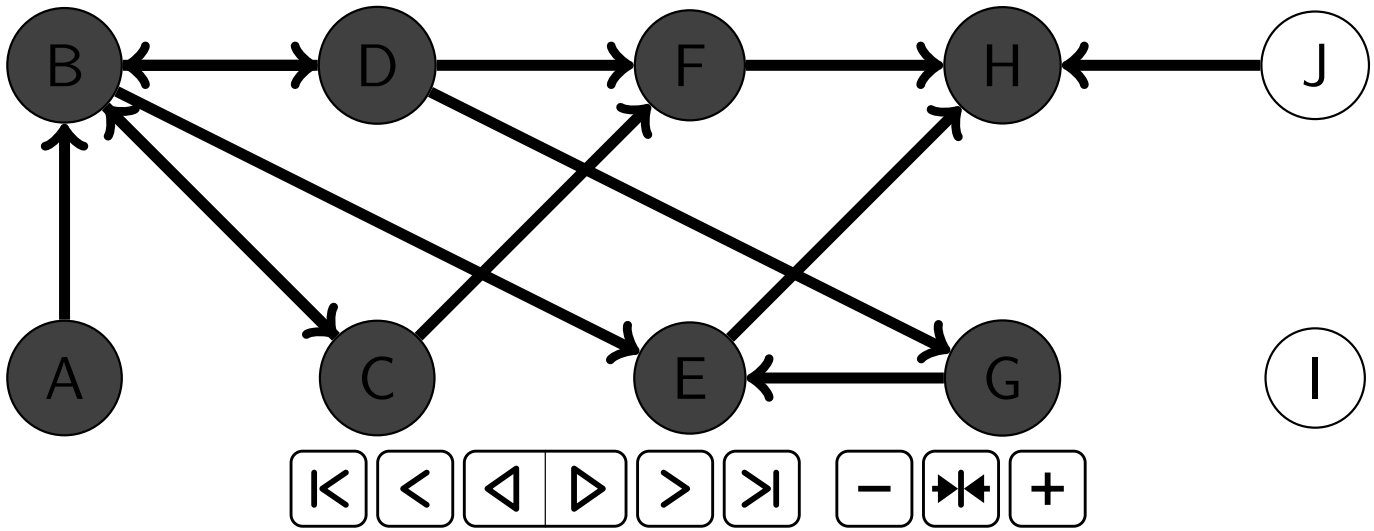
- on crée une liste vide qui contiendra la liste des nœuds accessibles depuis le nœud de départ
- on crée une liste booléenne indiquant pour chaque nœud s'il a déjà été visité
- on crée une pile ou une file contenant le nœud de départ
- à chaque étape, dépile/défile un élément (qu'on ajoute aux nœuds accessibles) et on empile/enfile tous les voisins non encore explorés de cet élément
- on s'arrête lorsque la pile/file est vide

Parcours en largeur

Le principe est le suivant :

- on part du nœud de départ et on liste tous ses voisins, nommés « premiers voisins » ;
- pour chacun des « premier voisins » pris dans l'ordre, on liste les voisins non encore découverts ; ce sont les « second voisins » ;
- on continue tant qu'on continue à découvrir des voisins.

Exemple de parcours en largeur



= nœud complètement exploré

= nœud en cours d'exploration

= nœud découvert, non exploré

= nouveaux nœuds découverts

= nœud non découvert

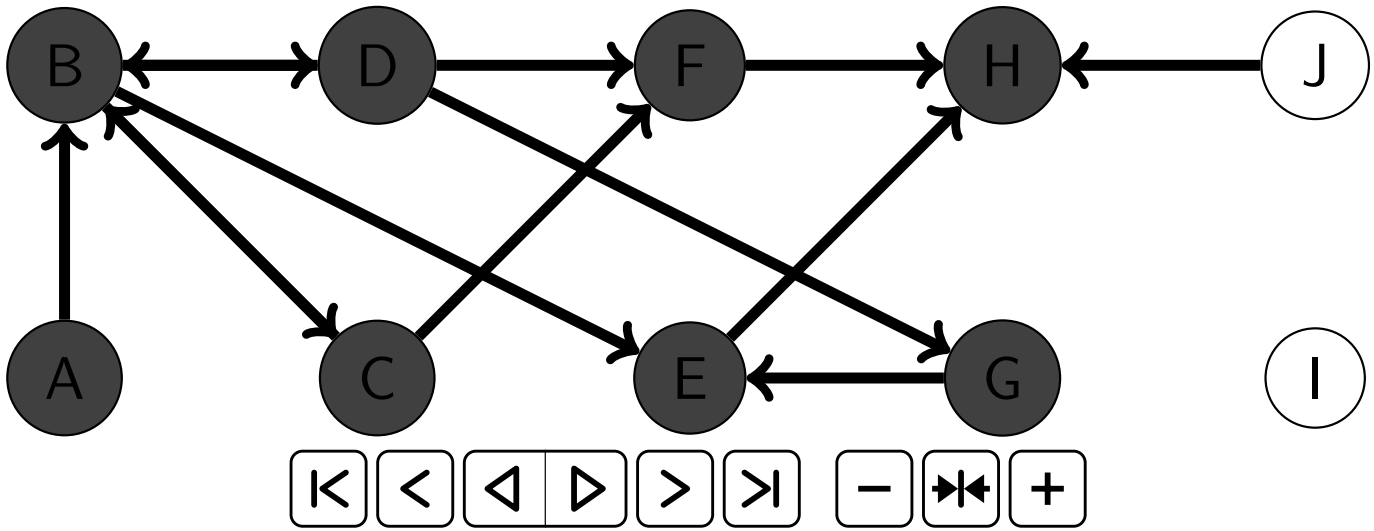
Parcours en profondeur

Le principe est le suivant :

- on part du nœud de départ, et on choisit le premier cœud dans la liste de ses voisins ;
- on part de ce nœud, et on recommence : on choisit le premier nœud non exploré dans sa liste de voisins ;
- on continue jusqu'à arriver à un nœud qui n'a aucun voisins non exploré : on recule alors jusqu'à trouver un nœud qui a des voisins inexplorés ;
- on continue jusqu'à ce qu'en reculant on revienne au nœud de départ et que tous ses voisins aient été explorés.

On peut aussi le décrire sous une **forme récursive** : « pour chaque voisin du nœud de départ, on effectue un parcours en profondeur à partir de ce voisin. »

Exemple de parcours en profondeur



= nœud complètement exploré

= nœud en cours d'exploration

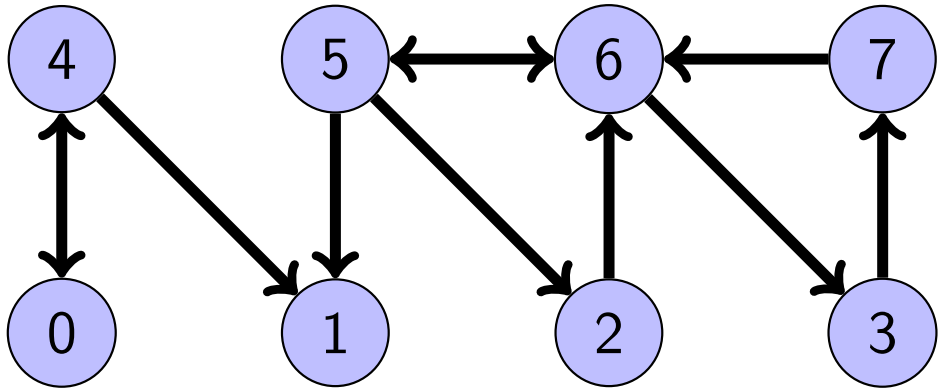
= nœud découvert, exploration en cours

= nouveaux nœuds découverts

= nœud non découvert

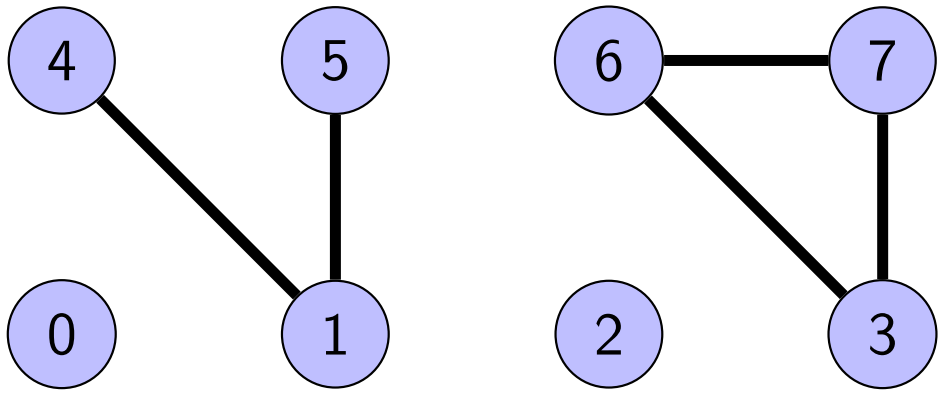
Parcours : exemple

Parcourir en profondeur puis en largeur depuis 2 dans ce graphe



Application : détermination des composantes connexes

Dans le cas d'un graphe non orienté, le parcours en profondeur permet de déterminer les composantes connexes :



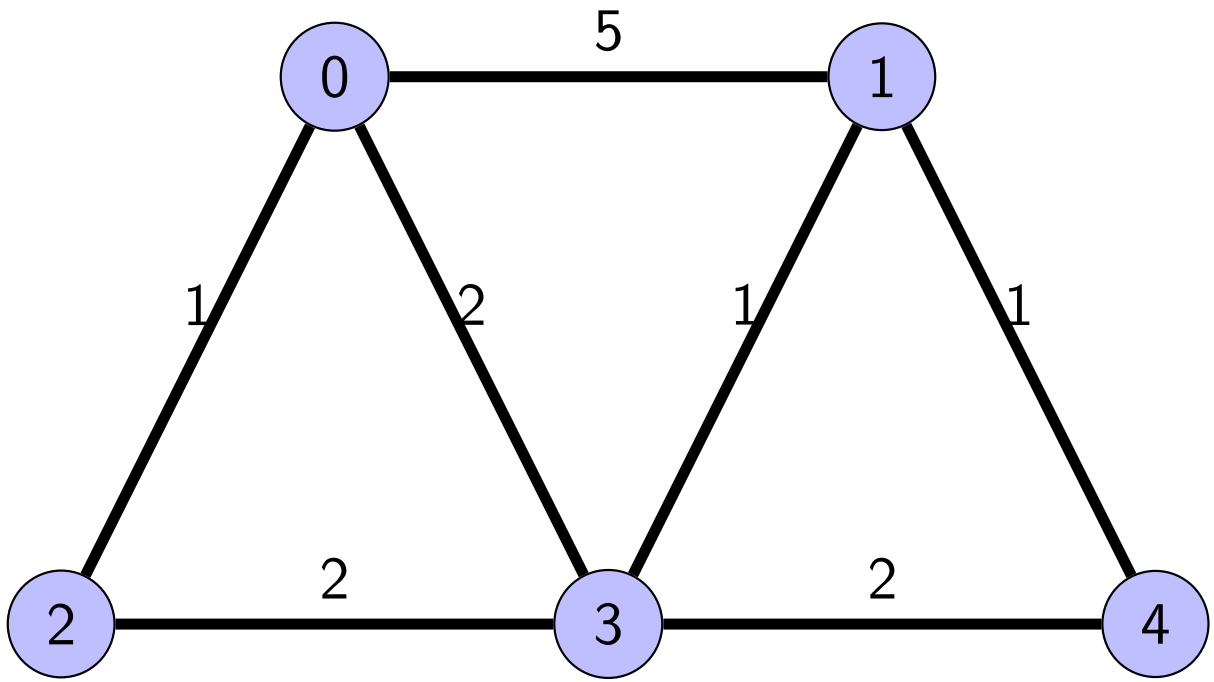
Section 3

Recherche de plus court chemin

Algorithme de Dijkstra

On considère :

- Un graphe G non orienté
- Représenté par listes d'adjacence
- Avec des poids **positifs** sur les arêtes



l'objectif est de déterminer le temps minimal pour atteindre les autres sommets à partir d'un sommet x_0 de départ

Algorithme de Dijkstra : principe

L'algorithme de Dijkstra est un **algorithme glouton**. Il utilise 3 listes :

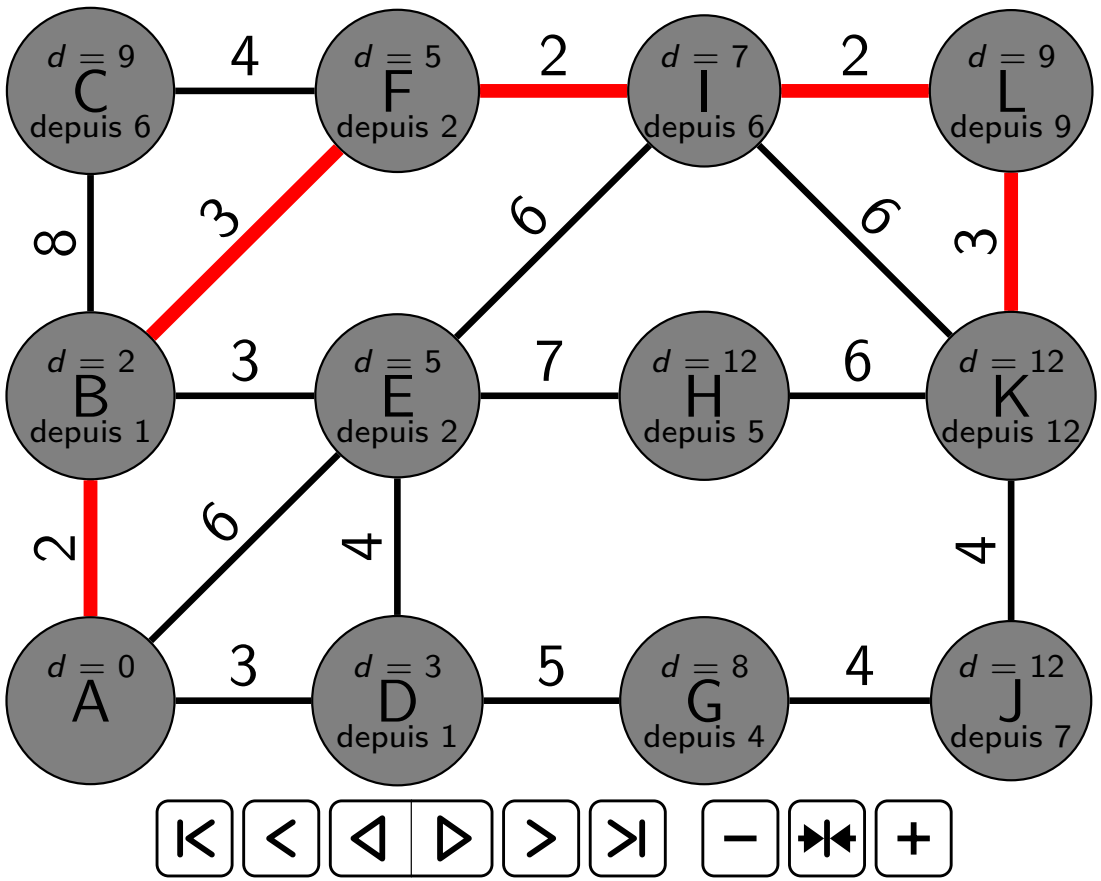
- une liste contenant les nœuds déjà traités ; initialement, elle est vide ;
- une liste d contenant le coût pour atteindre chaque nœud depuis x_0. Initialement $d(x_0) = 0$ et $d(x) = +\infty$ pour les autres sommets ;
- une liste contenant, pour chaque nœud, le nœud par lequel arrive le chemin de moindre coût (liste des provenances).

Algorithme de Dijkstra : principe

Il fonctionne selon une boucle :

- à chaque itération on choisit parmi les nœuds non traités le nœud y le moins coûteux à atteindre depuis x_0 (stratégie gloutonne) ; notons-le x_s ;
- on met alors à jour les coûts et les chemins en fonction des arêtes de y (on parle de relâchement d'arête) : pour chaque voisin v de x_s, on calcule si le coût de x_0 à x_s plus le coût de x_s à v est inférieur au coût déjà calculé de x_0 à v ; si c'est le cas, le chemin $x_0 \to x_s \to v$ devient le nouveau chemin préférentiel de x_0 à v ;
- on arrête lorsque le sommet le moins coûteux à ajouter est le sommet où on souhaite aller.

Exemple : recherche du chemin de A à K

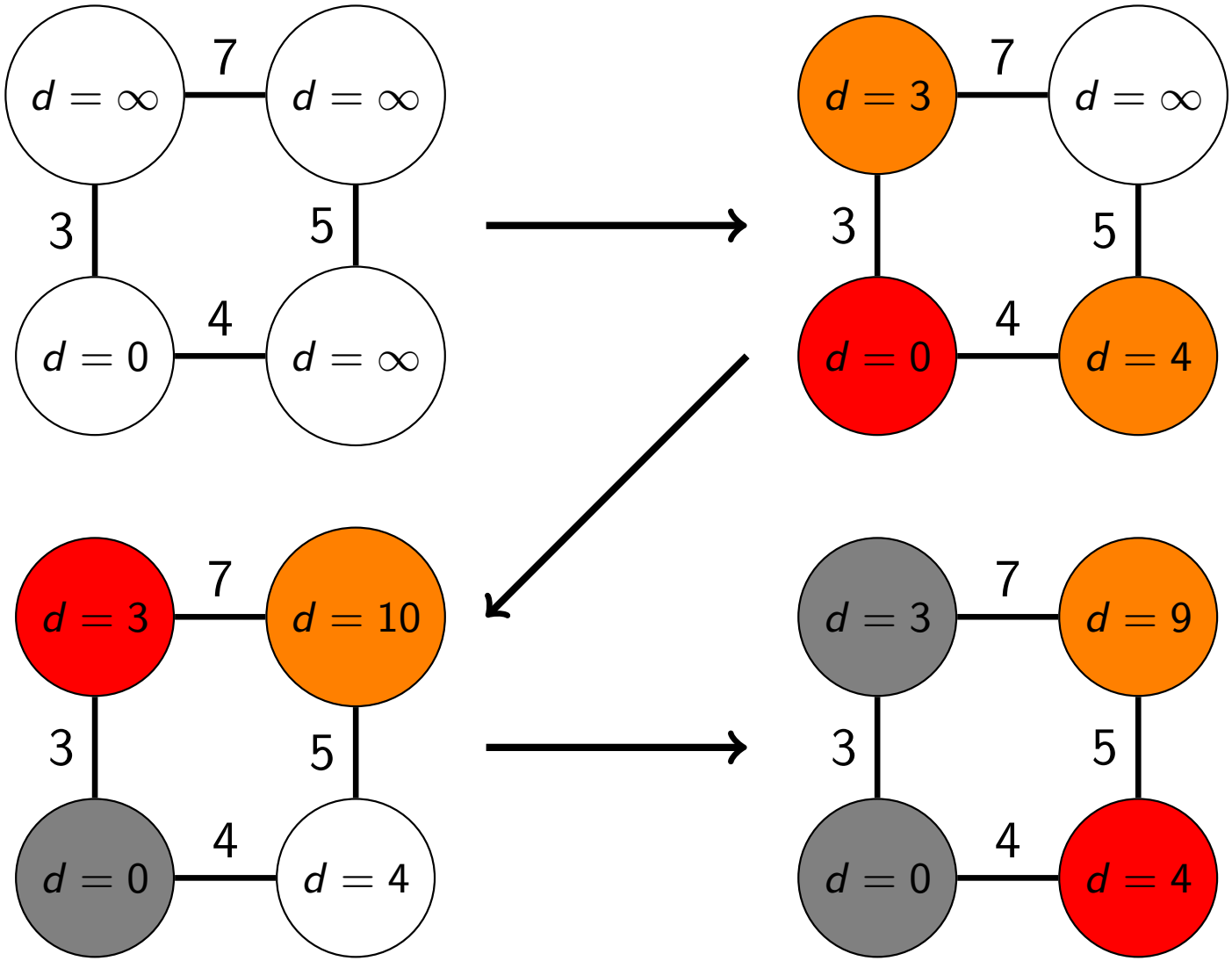


= nœud déjà exploré

= nœud dont on est en train de relâcher les arrêtes

= nœud dont on met à jour la distance

L'algorithme de Dijkstra est-il optimal ?



Algorithme de Dijkstra : complexité

On admet que :

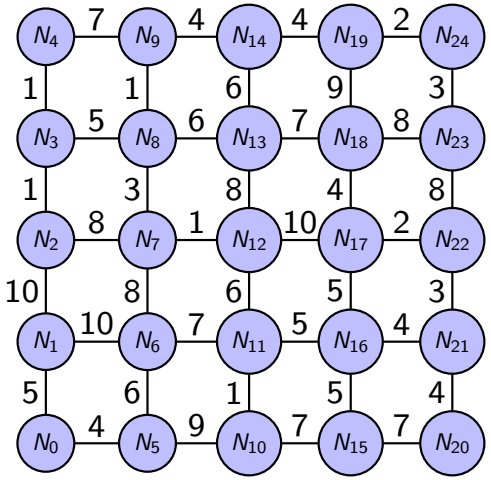
Complexité temporelle de Dijkstra)

Les implémentations les plus efficaces de l'algorithme de Dijkstra calculent un plus court chemin depuis un sommet x_0 vers tous les autres sommets en temps $O((n+m)\log n)$.

La complexité de cet algorithme est mauvaise car il recherche tous les chemins possibles autour du nœud de départ. Quand on connaît un peu la structure du graphe, il existe des possibilités pour trouver plus vite le bon chemin.

Amélioration : l'algorithme A*

Considérons un graphe dont on connaît les poids des chemins mais aussi les positions des nœuds sur une carte :



Si on souhaite aller du nœud N_8 au nœud N_{15}, commencer par aller vers N_9 n'est pas forcément pertinent.

Algorithme A*

L'algorithme $A*$ fonctionne comme celui de Dijkstra, sauf que le nœud dont on relâche les arrêtes est le nœud i qui minimise la somme $distance(debut, i) + \left\| N_i \vec{N}_{fin} \right\|$