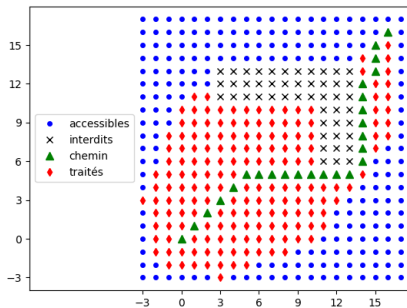


Graphes en Python

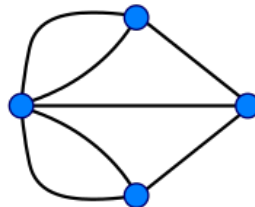
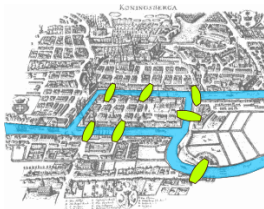
Jules Svartz

Lycée Masséna



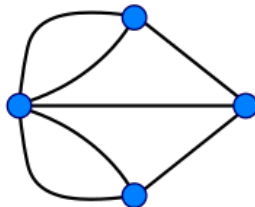
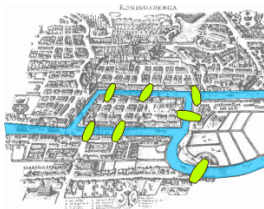
Historique

Graphes : début au 18ème siècle, ballade d'Euler dans Königsberg



Historique

Graphes : début au 18ème siècle, ballade d'Euler dans Königsberg

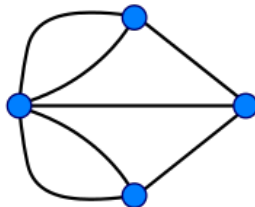
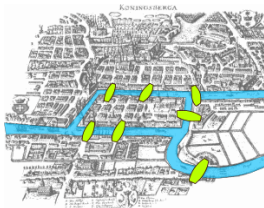


Applications :

- graphe du web (plus de 8 milliards d'URL valides)

Historique

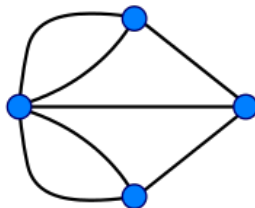
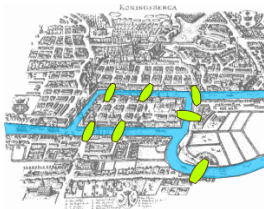
Graphes : début au 18ème siècle, ballade d'Euler dans Königsberg



Applications :

- graphe du web (plus de 8 milliards d'URL valides)
- graphes des réseaux sociaux, ex : Facebook $\simeq$ 2.91 milliards d'utilisateurs.

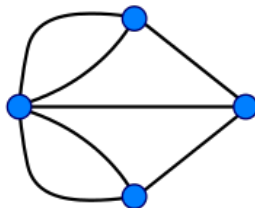
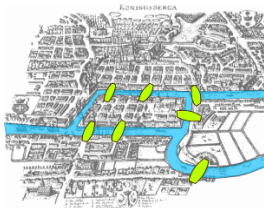
Graphes : début au 18ème siècle, ballade d'Euler dans Königsberg



Applications :

- graphe du web (plus de 8 milliards d'URL valides)
- graphes des réseaux sociaux, ex : Facebook $\simeq$ 2.91 milliards d'utilisateurs.
- graphe des routes en France (35000 communes françaises, nombre de routes ?)

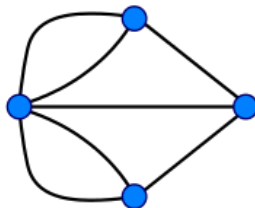
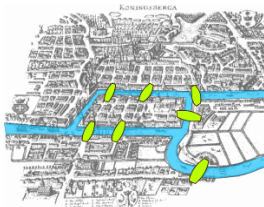
Graphes : début au 18ème siècle, ballade d'Euler dans Königsberg



Applications :

- graphe du web (plus de 8 milliards d'URL valides)
- graphes des réseaux sociaux, ex : Facebook $\simeq$ 2.91 milliards d'utilisateurs.
- graphe des routes en France (35000 communes françaises, nombre de routes ?)
- vols en avion dans le monde ($\simeq$ 100000 vols commerciaux par jour)

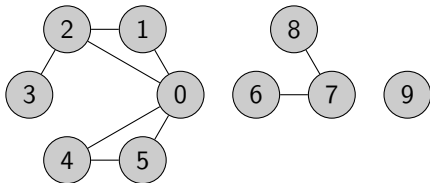
Graphes : début au 18ème siècle, ballade d'Euler dans Königsberg



Applications :

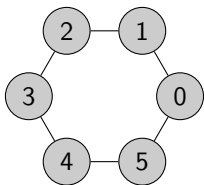
- graphe du web (plus de 8 milliards d'URL valides)
- graphes des réseaux sociaux, ex : Facebook $\simeq$ 2.91 milliards d'utilisateurs.
- graphe des routes en France (35000 communes françaises, nombre de routes ?)
- vols en avion dans le monde ($\simeq$ 100000 vols commerciaux par jour)
- réseaux de distribution eau/électricité...
- etc...

Vocabulaire : graphes non orientés

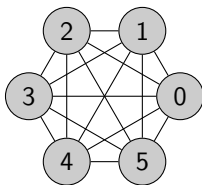


- Graphe non orienté : couple $G = (V, E)$: *noeuds* (ou *sommets*), *arêtes*
- *boucle*, *multi-arête*
- *sommets voisins*, *arête incidente*, *degré* d'un sommet.
- *chemin*, *cycle*.
- *connexité*, *composante connexe*.
- *arbre*.

Exemple de graphes non orientés

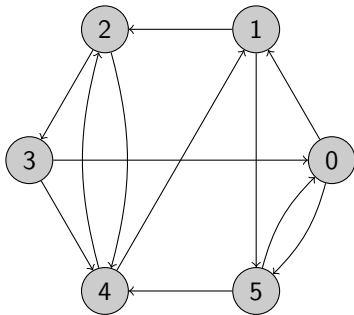


Cycle C_6

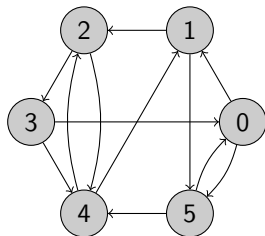


Clique K_6

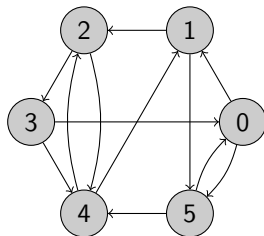
Vocabulaire : graphes orientés



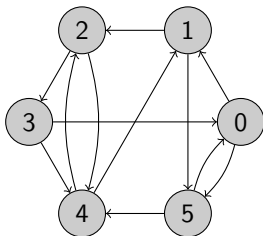
- Graphe orienté : couple $G = (V, E)$: *noeuds* (ou *sommets*), *arcs*
- *degré entrant*, *degré sortant*
- *circuit*.
- *composante fortement connexe*.



Implémentation



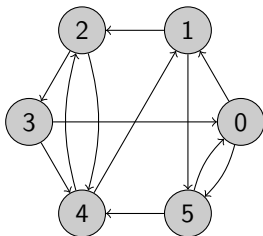
- Repr. creuse : listes d'adjacence (dictionnaire ok)
 $G = [[5, 1], [2, 5], [3], [0, 4], [2, 1], [0, 4]]$



- Repr. creuse : listes d'adjacence (dictionnaire ok)
 $G = [[5, 1], [2, 5], [3], [0, 4], [2, 1], [0, 4]]$
- Repr. dense : matrice d'adjacence.

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad M = \begin{bmatrix} [0, 1, 0, 0, 0, 1], \\ [0, 0, 1, 0, 0, 1], \\ [0, 0, 0, 1, 1, 0], \\ [1, 0, 0, 0, 1, 0], \\ [0, 1, 1, 0, 0, 0], \\ [1, 0, 0, 0, 1, 0] \end{bmatrix}$$

Implémentation



- Repr. creuse : listes d'adjacence (dictionnaire ok)
 $G = [[5, 1], [2, 5], [3], [0, 4], [2, 1], [0, 4]]$
- Repr. dense : matrice d'adjacence.

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad M = \begin{bmatrix} [0, 1, 0, 0, 0, 1], \\ [0, 0, 1, 0, 0, 1], \\ [0, 0, 0, 1, 1, 0], \\ [1, 0, 0, 0, 1, 0], \\ [0, 1, 1, 0, 0, 0], \\ [1, 0, 0, 0, 1, 0] \end{bmatrix}$$

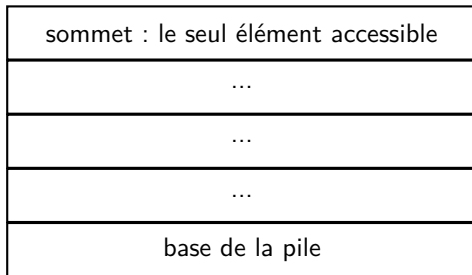
- Graphe non orienté : arête $\{x, y\} \iff$ arcs $x \rightarrow y$ et $y \rightarrow x$.

Intermède : structure de pile

Pile : principe *LIFO*

sommet : le seul élément accessible
...
...
...
base de la pile

Pile : principe *LIFO*



Opérations :

- Création d'une pile vide ;
- Test si une pile est vide ;
- Rajout d'un élément au sommet d'une pile ;
- Accès au sommet d'une pile non vide ;
- Suppression (et renvoi) du sommet d'une pile non vide.

- liste classique via `append`, `pop` et accès au dernier élément.

```
>>> P=[]
>>> P.append(3)
>>> P.append(5)
>>> P.pop()
5
>>> P[-1]
3
>>> P==[]
False
```

- Module *deque* (voir la suite).

File : principe *FIFO*

Opérations :

- Création d'une file vide ;
- Test si une file est vide ;
- Rajout d'un élément dans la file ;
- Suppression (et renvoi) du premier élément inséré dans la file.

- liste classique via `append`, `pop(0)` ?

```
>>> F = []  
>>> F.append(6)  
>>> F.append(4)  
>>> F.pop(0)  
6  
>>> F.pop(0)  
4  
>>> F == []  
True
```

Mauvais en complexité !

- Utilisation de deux listes (deux piles !)
- Module *deque*.

Structure de pile et de file en Python : module deque

```
• >>> 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) ; f #ajout à gauche
deque([5, 0, 1, 2, 3])
>>> f.popleft() #suppression à gauche
5
>>> f
deque([0, 1, 2, 3])
>>> len(f)
4
```

- pile via `append` et `pop` (ou `appendleft` et `popleft`)
- file via `append` et `popleft` (ou `appendleft` et `pop`)

Algorithme 1 : Parcours générique de graphe

Entrée : Un graphe G donné par listes d'adjacence, un sommet de départ s_0

$a_traiter \leftarrow \{s_0\};$

$B \leftarrow [\text{Faux}, \dots, \text{Faux}] ;$

$B[s_0] \leftarrow \text{Vrai};$

tant que $a_traiter$ est non vide **faire**

$s \leftarrow$ sortir un élément de $a_traiter$;

pour tout voisin s' de s tel que $B[s']$ est *Faux* **faire**

$a_traiter \leftarrow a_traiter \cup \{s'\};$

$B[s'] \leftarrow \text{Vrai}$

Questions :

Algorithme 1 : Parcours générique de graphe

Entrée : Un graphe G donné par listes d'adjacence, un sommet de départ s_0

$a_traiter \leftarrow \{s_0\};$

$B \leftarrow [\text{Faux}, \dots, \text{Faux}] ;$

$B[s_0] \leftarrow \text{Vrai};$

tant que $a_traiter$ est non vide **faire**

$s \leftarrow$ sortir un élément de $a_traiter$;

pour tout voisin s' de s tel que $B[s']$ est *Faux* **faire**

$a_traiter \leftarrow a_traiter \cup \{s'\};$

$B[s'] \leftarrow \text{Vrai}$

Questions :

- Terminaison ?

Algorithme 1 : Parcours générique de graphe

Entrée : Un graphe G donné par listes d'adjacence, un sommet de départ s_0

$a_traiter \leftarrow \{s_0\};$

$B \leftarrow [\text{Faux}, \dots, \text{Faux}] ;$

$B[s_0] \leftarrow \text{Vrai};$

tant que $a_traiter$ est non vide **faire**

$s \leftarrow$ sortir un élément de $a_traiter$;

pour tout voisin s' de s tel que $B[s']$ est *Faux* **faire**

$a_traiter \leftarrow a_traiter \cup \{s'\};$

$B[s'] \leftarrow \text{Vrai}$

Questions :

- Terminaison ?
- Sommets atteints ?

Algorithme 1 : Parcours générique de graphe

Entrée : Un graphe G donné par listes d'adjacence, un sommet de départ s_0

$a_traiter \leftarrow \{s_0\};$

$B \leftarrow [\text{Faux}, \dots, \text{Faux}] ;$

$B[s_0] \leftarrow \text{Vrai};$

tant que $a_traiter$ est non vide **faire**

$s \leftarrow$ sortir un élément de $a_traiter$;

pour tout voisin s' de s tel que $B[s']$ est *Faux* **faire**

$a_traiter \leftarrow a_traiter \cup \{s'\};$

$B[s'] \leftarrow \text{Vrai}$

Questions :

- Terminaison ?
- Sommets atteints ?
- Complexité ?

Algorithme 1 : Parcours générique de graphe

Entrée : Un graphe G donné par listes d'adjacence, un sommet de départ s_0

$a_traiter \leftarrow \{s_0\};$

$B \leftarrow [\text{Faux}, \dots, \text{Faux}] ;$

$B[s_0] \leftarrow \text{Vrai};$

tant que $a_traiter$ est non vide **faire**

$s \leftarrow$ sortir un élément de $a_traiter$;

pour tout voisin s' de s tel que $B[s']$ est *Faux* **faire**

$a_traiter \leftarrow a_traiter \cup \{s'\};$

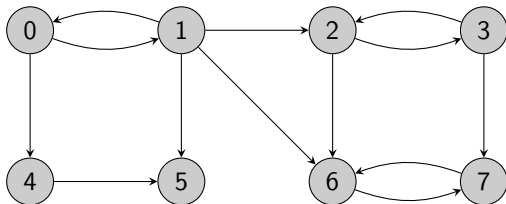
$B[s'] \leftarrow \text{Vrai}$

Questions :

- Terminaison ?
- Sommets atteints ?
- Complexité ?
- Applications ?

- Utilisation d'une file pour `a_traiter`.
- Calcul des distances depuis s_0 .
- Rajouter `pred` pour des calculs de plus courts chemins.

- Utilisation d'une file pour `a_traiter`.
- Calcul des distances depuis s_0 .
- Rajouter `pred` pour des calculs de plus courts chemins.
- Exemple :



Parcours en profondeur (complet)

- Discussion pile / récursivité
- Mutualisation liste de booléens

Algorithme 2 : Parcours en profondeur complet du graphe

Entrée : Un graphe G donné par liste d'adjacence

$\text{deja_vu}[v] \leftarrow \text{Faux}$ pour tout sommet v ;

Fonction $pp(u)$:

$\text{deja_vu}[u] \leftarrow \text{Vrai}$;

pour *tout voisin v de u* **faire**

si $\text{deja_vu}[v] = \text{Faux}$ **alors**

$pp(v)$

pour *tout sommet u du graphe* **faire**

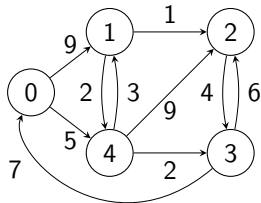
si $\text{deja_vu}[u] = \text{Faux}$ **alors**

$pp(u)$

- Calcul des composantes connexes (code).
- Existence d'un circuit dans un graphe orienté : trois couleurs blanc / gris / noir
 $\leadsto$ compilation de programme
- Existence d'un cycle dans un graphe non orienté : **attention** $u \rightarrow v$ et $v \rightarrow u$
 $\leadsto$ supprimer des arêtes pour garder un graphe connexe ?
- Tri topologique d'un graphe orienté sans circuit
 $\leadsto$ compilation de programme.
- Calcul des composantes fortement connexes d'un graphe orienté.

Graphes pondérés

- fonction de pondération : $\omega : E \rightarrow \mathbb{R}$
- Modification de la représentation :
 - listes d'adjacence : couple $(v, \omega(u, v))$.
 - matrice d'adjacence : $a_{i,j} = \omega(i, j)$ avec convention 0 si $i = j$ et $+\infty$ si l'arc n'existe pas.



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

```
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]]
```

```
(inf = float('inf'))
```

Plus courts chemins (i.e plus petits poids)

- poids d'un chemin : $c = s_0, \dots, s_n$, $\omega(c) = \sum_{i=0}^{n-1} \omega(s_i, s_{i+1})$

Plus courts chemins (i.e plus petits poids)

- poids d'un chemin : $c = s_0, \dots, s_n$, $\omega(c) = \sum_{i=0}^{n-1} \omega(s_i, s_{i+1})$
- $\delta(s, t) = \inf\{\omega(c) \mid c \text{ chemin de } s \text{ à } t\} \in \mathbb{R} \cup \{\pm\infty\}$. Supposer pas de circuit de poids < 0 .

Plus courts chemins (i.e plus petits poids)

- poids d'un chemin : $c = s_0, \dots, s_n$, $\omega(c) = \sum_{i=0}^{n-1} \omega(s_i, s_{i+1})$
- $\delta(s, t) = \inf\{\omega(c) \mid c \text{ chemin de } s \text{ à } t\} \in \mathbb{R} \cup \{\pm\infty\}$. Supposer pas de circuit de poids < 0 .
- Inégalité triangulaire : $\delta(s, t) \leq \delta(s, u) + \delta(u, t)$.

Plus courts chemins (i.e plus petits poids)

- poids d'un chemin : $c = s_0, \dots, s_n$, $\omega(c) = \sum_{i=0}^{n-1} \omega(s_i, s_{i+1})$
- $\delta(s, t) = \inf\{\omega(c) \mid c \text{ chemin de } s \text{ à } t\} \in \mathbb{R} \cup \{\pm\infty\}$. Supposer pas de circuit de poids < 0 .
- Inégalité triangulaire : $\delta(s, t) \leq \delta(s, u) + \delta(u, t)$.
- Corollaire : $\delta(s, t) \leq \delta(s, u) + \omega(u, t)$.

Plus courts chemins (i.e plus petits poids)

- poids d'un chemin : $c = s_0, \dots, s_n$, $\omega(c) = \sum_{i=0}^{n-1} \omega(s_i, s_{i+1})$
- $\delta(s, t) = \inf\{\omega(c) \mid c \text{ chemin de } s \text{ à } t\} \in \mathbb{R} \cup \{\pm\infty\}$. Supposer pas de circuit de poids < 0 .
- Inégalité triangulaire : $\delta(s, t) \leq \delta(s, u) + \delta(u, t)$.
- Corollaire : $\delta(s, t) \leq \delta(s, u) + \omega(u, t)$.
- Prop : optimalité des sous-problèmes dans les calculs de plus courts chemins.

- But : source s , calculer tous les $(\delta(s, t))_{t \in V}$ (généralise le parcours en largeur).

- But : source s , calculer tous les $(\delta(s, t))_{t \in V}$ (généralise le parcours en largeur).
- Contrainte : poids positifs !

- But : source s , calculer tous les $(\delta(s, t))_{t \in V}$ (généralise le parcours en largeur).
- **Contrainte : poids positifs !**
- Définitions utiles :
 - *Estimation des poids* depuis s : tableau d t.q $\forall t \ d[t] \geq \delta(s, t)$ pour tout sommet t .
 - *Relâchement de l'arc* $u \rightarrow v$: $d[v] \leftarrow \min(d[v], d[u] + \omega(u, v))$.

- But : source s , calculer tous les $(\delta(s, t))_{t \in V}$ (généralise le parcours en largeur).
- **Contrainte : poids positifs !**
- Définitions utiles :
 - *Estimation des poids* depuis s : tableau d t.q $\forall t \ d[t] \geq \delta(s, t)$ pour tout sommet t .
 - *Relâchement de l'arc* $u \rightarrow v$: $d[v] \leftarrow \min(d[v], d[u] + \omega(u, v))$.
- Prop : relâcher des arcs sur une estimation de poids donne une estimation de poids.

Algorithme 3 : Algorithme de Dijkstra

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par listes d'adjacence, avec $\omega(E) \subset \mathbb{R}_+$, un sommet s

Sortie : Les poids minimaux $\delta(s, t)$ pour tout $t \in V$

$d[t] \leftarrow +\infty$ pour tout $t \in V$; $d[s] \leftarrow 0$;

$H \leftarrow \emptyset$; $F \leftarrow \{s\}$;

tant que $F \neq \emptyset$ **faire**

$u \leftarrow$ Retirer de F un sommet v vérifiant $d[v]$ minimal parmi les sommets de F ;

pour *tout* voisin v de u **faire**

si v n'est ni dans F ni dans H **alors**

 Ajouter v à F

$d[v] \leftarrow \min(d[v], d[u] + \omega(u, v))$

 Ajouter u à H

Renvoyer d

Questions :

Algorithme 3 : Algorithme de Dijkstra

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par listes d'adjacence, avec $\omega(E) \subset \mathbb{R}_+$, un sommet s

Sortie : Les poids minimaux $\delta(s, t)$ pour tout $t \in V$

$d[t] \leftarrow +\infty$ pour tout $t \in V$; $d[s] \leftarrow 0$;

$H \leftarrow \emptyset$; $F \leftarrow \{s\}$;

tant que $F \neq \emptyset$ **faire**

$u \leftarrow$ Retirer de F un sommet v vérifiant $d[v]$ minimal parmi les sommets de F ;

pour *tout* voisin v de u **faire**

si v n'est ni dans F ni dans H **alors**

 Ajouter v à F

$d[v] \leftarrow \min(d[v], d[u] + \omega(u, v))$

 Ajouter u à H

Renvoyer d

Questions :

- parcours générique $+$...

Algorithme 3 : Algorithme de Dijkstra

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par listes d'adjacence, avec $\omega(E) \subset \mathbb{R}_+$, un sommet s

Sortie : Les poids minimaux $\delta(s, t)$ pour tout $t \in V$

$d[t] \leftarrow +\infty$ pour tout $t \in V$; $d[s] \leftarrow 0$;

$H \leftarrow \emptyset$; $F \leftarrow \{s\}$;

tant que $F \neq \emptyset$ **faire**

$u \leftarrow$ Retirer de F un sommet v vérifiant $d[v]$ minimal parmi les sommets de F ;

pour *tout* voisin v de u **faire**

si v n'est ni dans F ni dans H **alors**

 Ajouter v à F

$d[v] \leftarrow \min(d[v], d[u] + \omega(u, v))$

 Ajouter u à H

Renvoyer d

Questions :

- parcours générique $+$...
- estimation de poids à la fin $+$ correction.

Algorithme 3 : Algorithme de Dijkstra

Entrée : Un graphe pondéré $G = (V, E, \omega)$ donné par listes d'adjacence, avec $\omega(E) \subset \mathbb{R}_+$, un sommet s

Sortie : Les poids minimaux $\delta(s, t)$ pour tout $t \in V$

$d[t] \leftarrow +\infty$ pour tout $t \in V$; $d[s] \leftarrow 0$;

$H \leftarrow \emptyset$; $F \leftarrow \{s\}$;

tant que $F \neq \emptyset$ **faire**

$u \leftarrow$ Retirer de F un sommet v vérifiant $d[v]$ minimal parmi les sommets de F ;

pour *tout* voisin v de u **faire**

si v n'est ni dans F ni dans H **alors**

 Ajouter v à F

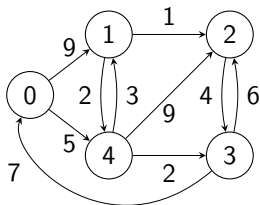
$d[v] \leftarrow \min(d[v], d[u] + \omega(u, v))$

 Ajouter u à H

Renvoyer d

Questions :

- parcours générique + ...
- estimation de poids à la fin + correction.
- Implémentation de F ?



- Exemple.
- Code pour un graphe représenté par matrice d'adjacence.

Algorithme A^*

- But : optimiser l'algorithme de Dijkstra lorsqu'on veut seulement $\delta(s, t)$, t fixé.

Algorithme A^*

- But : optimiser l'algorithme de Dijkstra lorsqu'on veut seulement $\delta(s, t)$, t fixé.
- Idée : s'arrêter dès que t traité. Peut-on faire mieux ?

Algorithme A^*

- But : optimiser l'algorithme de Dijkstra lorsqu'on veut seulement $\delta(s, t)$, t fixé.
- Idée : s'arrêter dès que t traité. Peut-on faire mieux ?
- *Heuristique* : une fonction $f : V \rightarrow \mathbb{R}_+$ estimant $\delta(u, t)$. Exemple ?

Algorithme A^*

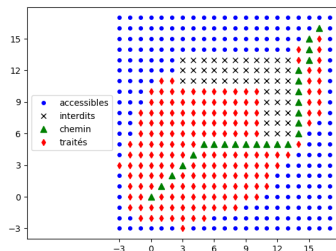
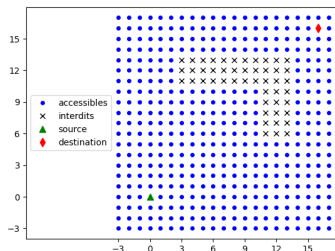
- But : optimiser l'algorithme de Dijkstra lorsqu'on veut seulement $\delta(s, t)$, t fixé.
- Idée : s'arrêter dès que t traité. Peut-on faire mieux ?
- *Heuristique* : une fonction $f : V \rightarrow \mathbb{R}_+$ estimant $\delta(u, t)$. Exemple ?
- *Heuristique admissible* : $0 \leq f(u) \leq \delta(u, t)$ pour tout u .

Algorithme A^*

- But : optimiser l'algorithme de Dijkstra lorsqu'on veut seulement $\delta(s, t)$, t fixé.
- Idée : s'arrêter dès que t traité. Peut-on faire mieux ?
- *Heuristique* : une fonction $f : V \rightarrow \mathbb{R}_+$ estimant $\delta(u, t)$. Exemple ?
- *Heuristique admissible* : $0 \leq f(u) \leq \delta(u, t)$ pour tout u .
- **Modification** : sortir un élément de F vérifiant $d[u] + f(u)$ minimal au lieu de $d[u]$ seulement.

Algorithme A^*

- But : optimiser l'algorithme de Dijkstra lorsqu'on veut seulement $\delta(s, t)$, t fixé.
- Idée : s'arrêter dès que t traité. Peut-on faire mieux ?
- *Heuristique* : une fonction $f : V \rightarrow \mathbb{R}_+$ estimant $\delta(u, t)$. Exemple ?
- *Heuristique admissible* : $0 \leq f(u) \leq \delta(u, t)$ pour tout u .
- **Modification** : sortir un élément de F vérifiant $d[u] + f(u)$ minimal au lieu de $d[u]$ seulement.



Comparaison : 196 nœuds traités au lieu de 2027.

Algorithme de Floyd-Warshall

- But : calcul de *toutes* les distances $\delta(s, t)$ dans le graphe, pour $s, t \in V$.
- Ne nécessite pas que les poids soient positifs (pas de circuit de poids < 0).
- Prop : il existe un plus court chemin qui ne passe que par des sommets distincts.
- Idée : $P_k = (p_{i,j}^k)_{(i,j) \in \llbracket 0, n-1 \rrbracket^2}$ avec $p_{i,j}^k = \inf \{ \omega(c) \mid c \text{ chemin de } i \text{ à } j, \text{ sommets intermédiaires dans } \llbracket 0, k-1 \rrbracket \}$
- Prop : $p_{i,j}^{k+1} = \min(p_{i,j}^k, p_{i,k}^k + p_{k,j}^k)$.
- Algo : calcul incrémental des $p_{i,j}^k$, on peut mettre à jour une unique matrice.
- Calcul en parallèle d'une *matrice de liaison* pour reconstruire des plus courts chemins.
- $O(n^3)$.