# Recherche de plus courts chemins

Ivan Noyer

Lycée Thiers

# Graphe pondéré

## Définition

On dit qu'un triplet $G = (V, E, p)$ est un graphe *pondéré* si le couple $(V, E)$ est un graphe et si $p$ est une fonction de $E$ dans $\mathbb{R}$.

- On dit que $p$ est la (fonction de) *pondération*. Souvent notée $w$ pour *weight*.

# Graphe pondéré

## Définition

On dit qu'un triplet $G = (V, E, p)$ est un graphe *pondéré* si le couple $(V, E)$ est un graphe et si $p$ est une fonction de $E$ dans $\mathbb{R}$.

- On dit que $p$ est la (fonction de) *pondération*. Souvent notée $w$ pour *weight*.
- Si $e \in E$, alors $p(e)$ est appelé *poids de* $e$.

# Graphe pondéré

## Définition

On dit qu'un triplet $G = (V, E, p)$ est un graphe *pondéré* si le couple $(V, E)$ est un graphe et si $p$ est une fonction de $E$ dans $\mathbb{R}$.

- On dit que $p$ est la (fonction de) *pondération*. Souvent notée $w$ pour *weight*.
- Si $e \in E$, alors $p(e)$ est appelé *poids de* $e$.
- Les graphes sont munis par défaut de la fonction de pondération $p : E \to \mathbb{R},\ e \mapsto 1$. Donc tout graphe est un graphe pondéré qui s'ignore.

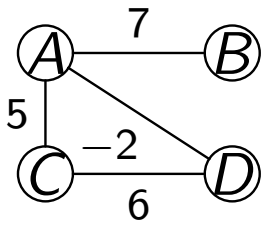
# Poids d'une chaîne

## Définition

Le *poids* d'une chaîne (resp. chemin) d'un graphe pondéré est la somme des poids des arcs qui le constituent.

Si $c$ est la chaîne $x_1 \dots x_n$ :

$$p(c) = \sum_{i=1}^{n-1} p(\{x_i, x_{i+1}\})$$




$$p(ACDA) = -2 + 6 + 5 = 9$$
$$p(ADC) = -2 + 6 = 4 \text{ et } p(AC) = 5$$
Le *plus court chemin* de $A$ à $C$ est $ADC$

# Plus court chemin

- Etant donné un graphe pondéré, on recherche le plus court chemin d'un sommet à un autre (ou à tous les autres, ou le plus court chemin entre tous les couples de sommets...)

# Plus court chemin

- Etant donné un graphe pondéré, on recherche le plus court chemin d'un sommet à un autre (ou à tous les autres, ou le plus court chemin entre tous les couples de sommets...)
- Exemple, dans une carte routière de la France, les villes étant des sommets, les routes étant des arcs, chercher le (ou les) trajet(s) de Paris à Mareille qui optimise(nt) l'un des critères suivants :

# Plus court chemin

- Etant donné un graphe pondéré, on recherche le plus court chemin d'un sommet à un autre (ou à tous les autres, ou le plus court chemin entre tous les couples de sommets...)
- Exemple, dans une carte routière de la France, les villes étant des sommets, les routes étant des arcs, chercher le (ou les) trajet(s) de Paris à Mareille qui optimise(nt) l'un des critères suivants :
  - la distance parcourue,

# Plus court chemin

- Etant donné un graphe pondéré, on recherche le plus court chemin d'un sommet à un autre (ou à tous les autres, ou le plus court chemin entre tous les couples de sommets...)
- Exemple, dans une carte routière de la France, les villes étant des sommets, les routes étant des arcs, chercher le (ou les) trajet(s) de Paris à Mareille qui optimise(nt) l'un des critères suivants :
  - la distance parcourue,
  - le temps mis,

# Plus court chemin

- Etant donné un graphe pondéré, on recherche le plus court chemin d'un sommet à un autre (ou à tous les autres, ou le plus court chemin entre tous les couples de sommets...)
- Exemple, dans une carte routière de la France, les villes étant des sommets, les routes étant des arcs, chercher le (ou les) trajet(s) de Paris à Mareille qui optimise(nt) l'un des critères suivants :
  - la distance parcourue,
  - le temps mis,
  - le prix des péages,

# Recherche de plus courts chemins

- Dans tout ce qui suit, les graphes sont considérés orientés. Si le graphe $G$ est non orienté, on l'oriente en construisant le graphe orienté tel que

# Recherche de plus courts chemins

- Dans tout ce qui suit, les graphes sont considérés orientés. Si le graphe $G$ est non orienté, on l'oriente en construisant le graphe orienté tel que
  - $G'$ a les mêmes sommets que $G$,

# Recherche de plus courts chemins

- Dans tout ce qui suit, les graphes sont considérés orientés. Si le graphe $G$ est non orienté, on l'oriente en construisant le graphe orienté tel que
  - $G'$ a les mêmes sommets que $G$,
  - pour tout arc $\{x, y\}$ de $G$, $G'$ possède les arcs $(x, y)$ et $(y, x)$

# Recherche de plus courts chemins

- Dans tout ce qui suit, les graphes sont considérés orientés. Si le graphe $G$ est non orienté, on l'oriente en construisant le graphe orienté tel que
  - $G'$ a les mêmes sommets que $G$,
  - pour tout arc $\{x, y\}$ de $G$, $G'$ possède les arcs $(x, y)$ et $(y, x)$
- On recherche pour tout couple de sommets $(i, j)$ le plus court chemin entre $i$ et $j$.

# Recherche de plus courts chemins

- Dans tout ce qui suit, les graphes sont considérés orientés. Si le graphe $G$ est non orienté, on l'oriente en construisant le graphe orienté tel que
  - $G'$ a les mêmes sommets que $G$,
  - pour tout arc $\{x, y\}$ de $G$, $G'$ possède les arcs $(x, y)$ et $(y, x)$
- On recherche pour tout couple de sommets $(i, j)$ le plus court chemin entre $i$ et $j$.
- Si $j$ n'est pas accessible depuis $i$, on pose que $d(i, j) = +\infty$.

# Longueur de plus courts chemins

## Cas particuliers des graphes orientés non pondérés

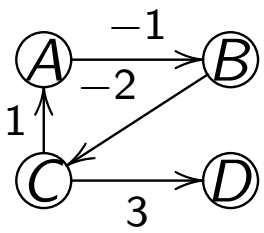
- Si le graphe $G$ est non pondéré, on considère sa pondération par défaut.

# Longueur de plus courts chemins

## Cas particuliers des graphes orientés non pondérés

- Si le graphe $G$ est non pondéré, on considère sa pondération par défaut.
- Lors du parcours en largeur, les sommets sont explorés par distance croissante au sommet source. Grâce à cette propriété on résout le problème de cheminement suivant : calculer les longueurs des plus courts chemins entre un sommet source et tous les sommets du graphe (voir TD).

# Circuits de poids négatif

- Pour rechercher un PCC dans un graphe, il faut s'assurer au préalable que celui-ci ne possède pas de circuit de poids négatif.
- Cela ne veut pas dire que le graphe ne peut pas contenir d'arcs de poids négatif.
- 


$$p(ABCA = -2)$$
$$p(ABCD) = 0$$
$$p(ABCABCD) = -2$$
$$p(ABCABCABCD) = -4$$

# Sous-chemin du plus court chemin

- Si un PCC de $A$ à $C$ passe par $B$, alors le sous-chemin entre $A$ et $B$ est un chemin de poids minimum.

# Sous-chemin du plus court chemin

- Si un PCC de $A$ à $C$ passe par $B$, alors le sous-chemin entre $A$ et $B$ est un chemin de poids minimum.
- En effet, s'il existe un autre chemin plus court entre $A$ et $B$, il suffit de le mettre à la place du premier pour obtenir un nouveau chemin de $A$ à $C$ encore plus court.

# Sous-chemin du plus court chemin

- Si un PCC de $A$ à $C$ passe par $B$, alors le sous-chemin entre $A$ et $B$ est un chemin de poids minimum.
- En effet, s'il existe un autre chemin plus court entre $A$ et $B$, il suffit de le mettre à la place du premier pour obtenir un nouveau chemin de $A$ à $C$ encore plus court.
- Ceci contredit le fait que le premier chemin avait un poids minimum.

# Sous-chemin du plus court chemin

- Si un PCC de $A$ à $C$ passe par $B$, alors le sous-chemin entre $A$ et $B$ est un chemin de poids minimum.
- En effet, s'il existe un autre chemin plus court entre $A$ et $B$, il suffit de le mettre à la place du premier pour obtenir un nouveau chemin de $A$ à $C$ encore plus court.
- Ceci contredit le fait que le premier chemin avait un poids minimum.
- Ceci est un cas particulier du *principe d'optimalité de Bellman* qui dit que l'on peut (dans certains cas) déduire une solution optimale d'un problème en combinant des solutions optimales d'une série de sous-problèmes.

# Sous-chemin du plus court chemin

- Si un PCC de $A$ à $C$ passe par $B$, alors le sous-chemin entre $A$ et $B$ est un chemin de poids minimum.
- En effet, s'il existe un autre chemin plus court entre $A$ et $B$, il suffit de le mettre à la place du premier pour obtenir un nouveau chemin de $A$ à $C$ encore plus court.
- Ceci contredit le fait que le premier chemin avait un poids minimum.
- Ceci est un cas particulier du *principe d'optimalité de Bellman* qui dit que l'on peut (dans certains cas) déduire une solution optimale d'un problème en combinant des solutions optimales d'une série de sous-problèmes.
- Pour les PCC, on l'utilise en calculant d'abord les PCC passant par un sous-ensemble de sommets avant de s'attaquer aux PCC passant par un ensemble de sommets plus gros.

# Présentation

- Calcule, dans un graphe orienté pondéré **par des réels positifs**, les plus courts chemins à partir d'une *source unique* en direction de *tous les autres sommets*.

# Présentation

- Calcule, dans un graphe orienté pondéré **par des réels positifs**, les plus courts chemins à partir d'une *source unique* en direction de *tous les autres sommets*.
- Dû à l'informaticien néerlandais Edsger Dijkstra (par ailleurs prix Turing),

# Présentation

- Calcule, dans un graphe orienté pondéré **par des réels positifs**, les plus courts chemins à partir d'une *source unique* en direction de *tous les autres sommets*.
- Dû à l'informaticien néerlandais Edsger Dijkstra (par ailleurs prix Turing),
- publié en 1959.

# Principe

- $G$ a pour ensemble de sommets $[\![1, n]\!]$.

1. Parfois aussi, on met à jour un tableau des *prédecesseurs*

# Principe

- $G$ a pour ensemble de sommets $[\![1, n]\!]$.
- On cherche les PCC depuis la *source* (ou *entrée*) $e$ vers tous les autres sommets.

1. Parfois aussi, on met à jour un tableau des *prédecesseurs*

# Principe

- $G$ a pour ensemble de sommets $[\![1, n]\!]$.
- On cherche les PCC depuis la *source* (ou *entrée*) $e$ vers tous les autres sommets.
- Principe :

1. Parfois aussi, on met à jour un tableau des *prédecesseurs*

# Principe

- $G$ a pour ensemble de sommets $[\![1, n]\!]$.
- On cherche les PCC depuis la *source* (ou *entrée*) $e$ vers tous les autres sommets.
- Principe :
  - A chaque tour, on choisit parmi tous les sommets verts celui dont la distance à l'entrée est minimale (le plus court chemin). Ce sommet devient rouge.

1. Parfois aussi, on met à jour un tableau des *prédecesseurs*

# Principe

- $G$ a pour ensemble de sommets $[\![1, n]\!]$.
- On cherche les PCC depuis la *source* (ou *entrée*) $e$ vers tous les autres sommets.
- Principe :
  - A chaque tour, on choisit parmi tous les sommets verts celui dont la distance à l'entrée est minimale (le plus court chemin). Ce sommet devient rouge.
  - Tout voisin bleu du sommet choisi est ajouté à l'ensemble des sommets verts (en ce sens c'est un parcours en largeur).
    Les informations concernant les distances à la source des voisins du sommet qui vient de devenir rouge sont mises à jour [1].

1. Parfois aussi, on met à jour un tableau des *prédecesseurs*

# Principe

- $G$ a pour ensemble de sommets $[\![1, n]\!]$.
- On cherche les PCC depuis la *source* (ou *entrée*) $e$ vers tous les autres sommets.
- Principe :
  - A chaque tour, on choisit parmi tous les sommets verts celui dont la distance à l'entrée est minimale (le plus court chemin). Ce sommet devient rouge.
  - Tout voisin bleu du sommet choisi est ajouté à l'ensemble des sommets verts (en ce sens c'est un parcours en largeur).
    Les informations concernant les distances à la source des voisins du sommet qui vient de devenir rouge sont mises à jour [1].
- C'est un algorithme glouton dont le résultat est optimal.
  Nous avons vu qu'il existe des algos gloutons non optimal (par exemple pour la coloration).

1. Parfois aussi, on met à jour un tableau des *prédecesseurs*

# Principe

L'ensemble des sommets verts est noté $F$ car il est souvent implémenté avec une file de priorité.

```
1  F:={e} /*sommets en cours de traitement (=verts)*/
2  E:=∅ /*sommets traités = sommets rouges*/
3  D:= tableau des distances minimales (d_e = 0, k ≠ e ⟹ d_k = ∞)
4  tant que F ≠ ∅ faire
5        choisir k ∈ F avec d_k minimal
6        F:=F \ {k}
7        E:=E ∪ {k} /* k devient rouge*/
8        pour tout voisin v de k non rouge faire
9              si v ∉ F : /*si v est bleu*/
10                F:=F ∪ {v} /*v devient vert*/
11             fin_si
12             si d_k + w(k → v) < d_v :
13                   /*passer par k pour atteindre v est plus rentable*/
14                   d_v:=d_k + w(k → v) /*maj tab. des distances*/
15             fin_si
16       fin_faire
17 fin_faire
```

# Terminaison

Pour un graphe $G = (S, A)$, on note $B = S \setminus (F \cup E)$ l'ensemble des sommets bleus.

- Variant : $|F| + |B|$ (cardinal de l'ensemble de sommets verts ou bleus). Quantité entière positive.

# Terminaison

Pour un graphe $G = (S, A)$, on note $B = S \setminus (F \cup E)$ l'ensemble des sommets bleus.

- Variant : $|F| + |B|$ (cardinal de l'ensemble de sommets verts ou bleus). Quantité entière positive.
- C'est une quantité décroissante strictement à la fin de chaque tour de boucle car :

# Terminaison

Pour un graphe $G = (S, A)$, on note $B = S \setminus (F \cup E)$ l'ensemble des sommets bleus.

- Variant : $|F| + |B|$ (cardinal de l'ensemble de sommets verts ou bleus). Quantité entière positive.
- C'est une quantité décroissante strictement à la fin de chaque tour de boucle car :
  - on retire toujours un sommet de $F$ (si ce n'est pas possible, $F = \emptyset$ et l'algorithme s'arrête).

# Terminaison

Pour un graphe $G = (S, A)$, on note $B = S \setminus (F \cup E)$ l'ensemble des sommets bleus.

- Variant : $|F| + |B|$ (cardinal de l'ensemble de sommets verts ou bleus). Quantité entière positive.
- C'est une quantité décroissante strictement à la fin de chaque tour de boucle car :
  - on retire toujours un sommet de $F$ (si ce n'est pas possible, $F = \emptyset$ et l'algorithme s'arrête).
  - tout sommet ajouté à $F$ est retiré de $B$.

# Terminaison

Pour un graphe $G = (S, A)$, on note $B = S \setminus (F \cup E)$ l'ensemble des sommets bleus.

- Variant : $|F| + |B|$ (cardinal de l'ensemble de sommets verts ou bleus). Quantité entière positive.
- C'est une quantité décroissante strictement à la fin de chaque tour de boucle car :
  - on retire toujours un sommet de $F$ (si ce n'est pas possible, $F = \emptyset$ et l'algorithme s'arrête).
  - tout sommet ajouté à $F$ est retiré de $B$.
  - ainsi, l'ensemble ds sommets verts ou bleus comporte un élément de moins à la fin d'un tour par rapport au tour précédent.

# Terminaison

Pour un graphe $G = (S, A)$, on note $B = S \setminus (F \cup E)$ l'ensemble des sommets bleus.

- Variant : $|F| + |B|$ (cardinal de l'ensemble de sommets verts ou bleus). Quantité entière positive.
- C'est une quantité décroissante strictement à la fin de chaque tour de boucle car :
  - on retire toujours un sommet de $F$ (si ce n'est pas possible, $F = \emptyset$ et l'algorithme s'arrête).
  - tout sommet ajouté à $F$ est retiré de $B$.
  - ainsi, l'ensemble ds sommets verts ou bleus comporte un élément de moins à la fin d'un tour par rapport au tour précédent.
- Puisqu'on a exhibé un variant : Terminaison OK.

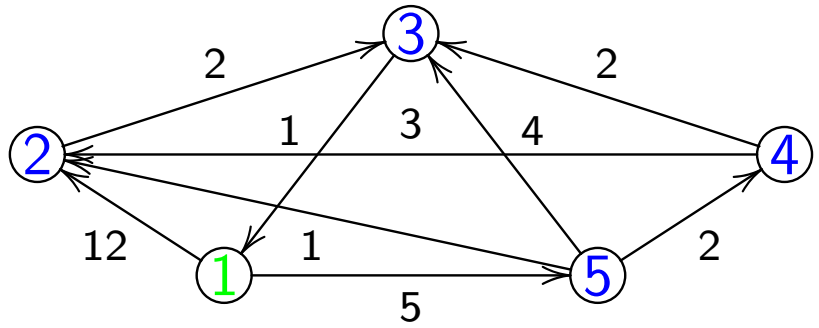
# Terminaison

Pour un graphe $G = (S, A)$, on note $B = S \setminus (F \cup E)$ l'ensemble des sommets bleus.

- Variant : $|F| + |B|$ (cardinal de l'ensemble de sommets verts ou bleus). Quantité entière positive.
- C'est une quantité décroissante strictement à la fin de chaque tour de boucle car :
  - on retire toujours un sommet de $F$ (si ce n'est pas possible, $F = \emptyset$ et l'algorithme s'arrête).
  - tout sommet ajouté à $F$ est retiré de $B$.
  - ainsi, l'ensemble ds sommets verts ou bleus comporte un élément de moins à la fin d'un tour par rapport au tour précédent.
- Puisqu'on a exhibé un variant : Terminaison OK.
- Correction : admise.

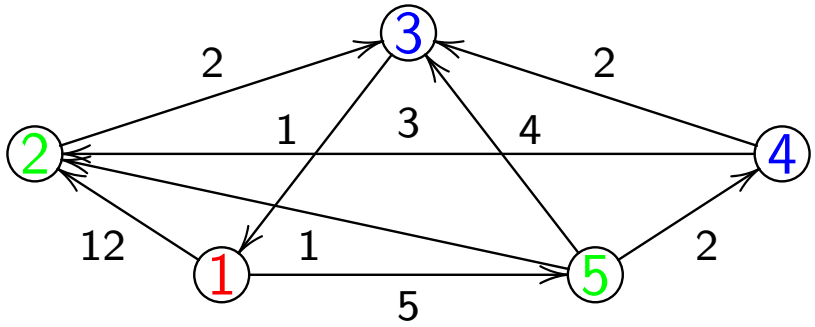
# Exemple

On gère en plus un tableau **P** (prédecesseur)qui garde en mémoire la dernière étape sur le chemin de la source au sommet considéré.



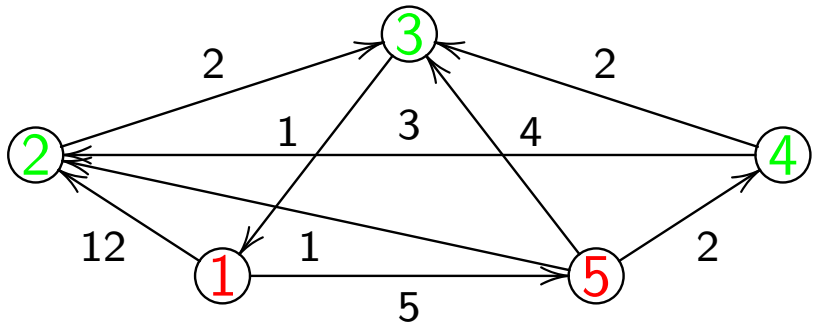

| F | E | T | P |
|---|---|---|---|
| $\{1\}$ | $\emptyset$ | $[\![0;\infty;\infty;\infty;\infty]\!]$ | $[\![1;-1;-1;-1;-1]\!]$ |

# Exemple



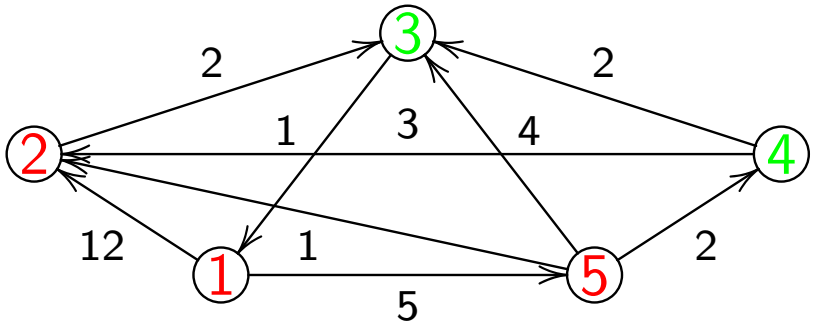

| F | E | T | P |
|---|---|---|---|
| $\{1\}$ | $\emptyset$ | $[\![0;\infty;\infty;\infty;\infty]\!]$ | $[\![1;-1;-1;-1;-1]\!]$ |
| $\{2;5\}$ | $\{1\}$ | $[\![0;12;\infty;\infty;5]\!]$ | $[\![1;1;-1;-1;1]\!]$ |

# Exemple



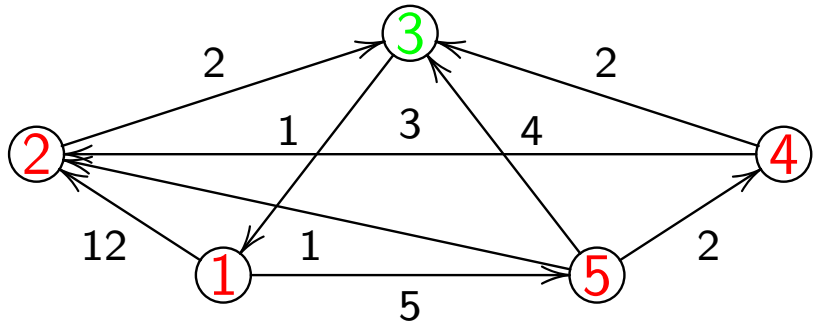

| F | E | T | P |
|---|---|---|---|
| $\{1\}$ | $\emptyset$ | $[\![0;\infty;\infty;\infty;\infty]\!]$ | $[\![1;-1;-1;-1;-1]\!]$ |
| $\{2;5\}$ | $\{1\}$ | $[\![0;12;\infty;\infty;5]\!]$ | $[\![1;1;-1;-1;1]\!]$ |
| $\{2;3;4\}$ | $\{1;5\}$ | $[\![0;6;9;7;5]\!]$ | $[\![1;5;5;5;1]\!]$ |

# Exemple



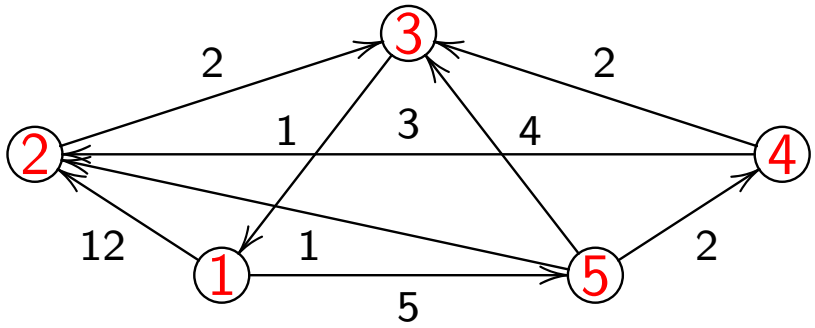

| F | E | T | P |
|---|---|---|---|
| $\{1\}$ | $\emptyset$ | $[\![0;\infty;\infty;\infty;\infty]\!]$ | $[\![1;-1;-1;-1;-1]\!]$ |
| $\{2;5\}$ | $\{1\}$ | $[\![0;12;\infty;\infty;5]\!]$ | $[\![1;1;-1;-1;1]\!]$ |
| $\{2;3;4\}$ | $\{1;5\}$ | $[\![0;6;9;7;5]\!]$ | $[\![1;5;5;5;1]\!]$ |
| $\{3;4\}$ | $\{1;5;2\}$ | $[\![0;6;8;7;5]\!]$ | $[\![1;5;2;5;1]\!]$ |

# Exemple



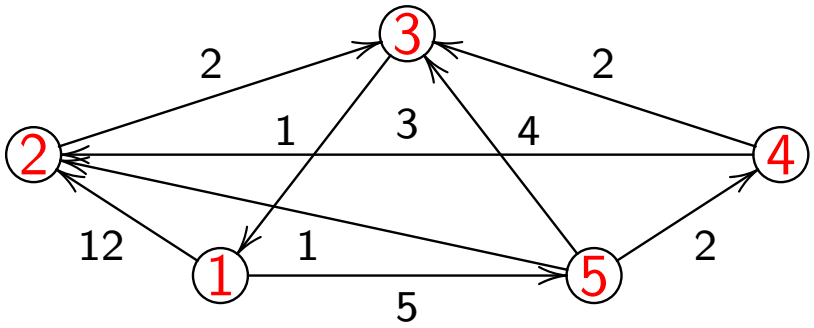

| F | E | T | P |
|---|---|---|---|
| $\{1\}$ | $\emptyset$ | $[\![0;\infty;\infty;\infty;\infty]\!]$ | $[\![1;-1;-1;-1;-1]\!]$ |
| $\{2;5\}$ | $\{1\}$ | $[\![0;12;\infty;\infty;5]\!]$ | $[\![1;1;-1;-1;1]\!]$ |
| $\{2;3;4\}$ | $\{1;5\}$ | $[\![0;6;9;7;5]\!]$ | $[\![1;5;5;5;1]\!]$ |
| $\{3;4\}$ | $\{1;5;2\}$ | $[\![0;6;8;7;5]\!]$ | $[\![1;5;2;5;1]\!]$ |
| $\{3\}$ | $\{1;5;2;4\}$ | $[\![0;6;8;7;5]\!]$ | $[\![1;5;2;5;1]\!]$ |

# Exemple




| F | E | T | P |
|---|---|---|---|
| $\{1\}$ | $\emptyset$ | $[\![0;\infty;\infty;\infty;\infty]\!]$ | $[\![1;-1;-1;-1;-1]\!]$ |
| $\{2;5\}$ | $\{1\}$ | $[\![0;12;\infty;\infty;5]\!]$ | $[\![1;1;-1;-1;1]\!]$ |
| $\{2;3;4\}$ | $\{1;5\}$ | $[\![0;6;9;7;5]\!]$ | $[\![1;5;5;5;1]\!]$ |
| $\{3;4\}$ | $\{1;5;2\}$ | $[\![0;6;f8;7;5]\!]$ | $[\![1;5;2;5;1]\!]$ |
| $\{3\}$ | $\{1;5;2;4\}$ | $[\![0;6;8;7;5]\!]$ | $[\![1;5;2;5;1]\!]$ |
| $\emptyset$ | $\{1;5;2;4;3\}$ | $[\![0;6;8;7;5]\!]$ | $[\![1;5;2;5;1]\!]$ |

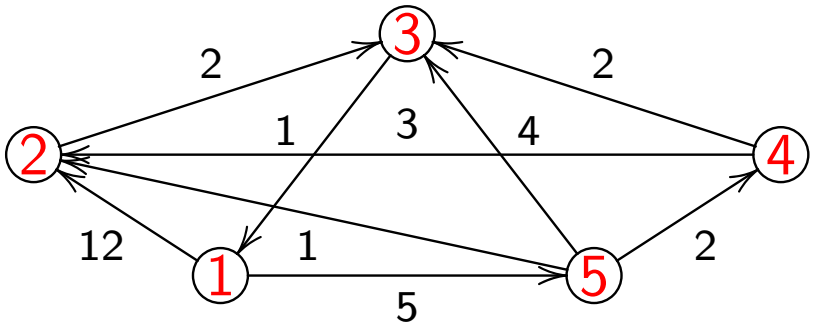
# Exemple




PCC pour aller de 1 à 3 :

- On a $T = [\![0; 6; 8; 7; 5]\!]$ et $P = [\![1; 5; 2; 5; 1]\!]$.

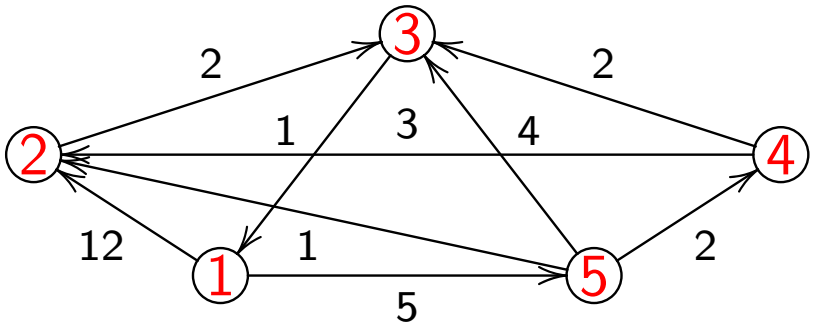
# Exemple




PCC pour aller de 1 à 3 :

- On a $T = [\![0; 6; 8; 7; 5]\!]$ et $P = [\![1; 5; 2; 5; 1]\!]$.
- prédecesseur de 3 : 2

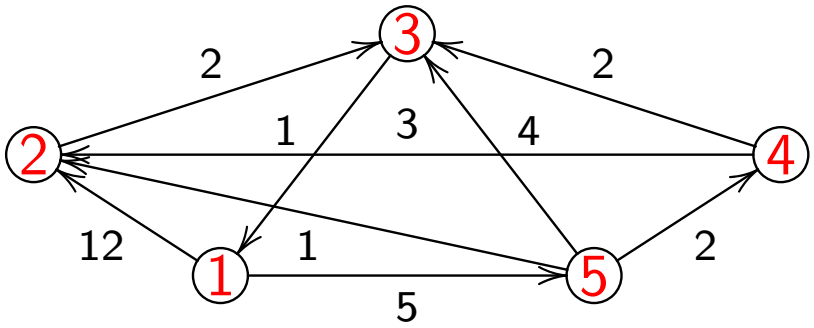
# Exemple




PCC pour aller de 1 à 3 :

- On a $T = [\![0; 6; 8; 7; 5]\!]$ et $P = [\![1; 5; 2; 5; 1]\!]$.
- prédecesseur de 3 : 2
- prédecesseur de 2 : 5

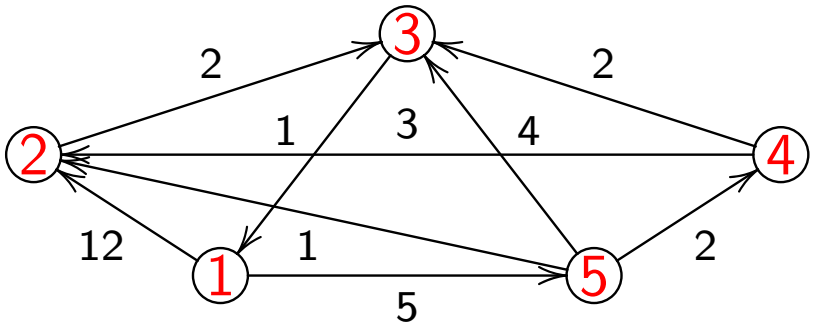
# Exemple




PCC pour aller de 1 à 3 :

- On a $T = [\![0; 6; 8; 7; 5]\!]$ et $P = [\![1; 5; 2; 5; 1]\!]$.
- prédecesseur de 3 : 2
- prédecesseur de 2 : 5
- prédecesseur de 5 : 1

# Exemple




PCC pour aller de 1 à 3 :

- On a $T = [\![0; 6; 8; 7; 5]\!]$ et $P = [\![1; 5; 2; 5; 1]\!]$.
- prédecesseur de 3 : 2
- prédecesseur de 2 : 5
- prédecesseur de 5 : 1
- donc : 1523 avec un coût de 8

# Complexité pour une source $e$

## Force brute pour un graphe de $n$ sommets et $p$ arcs

```
1  F:={e} ; E:=∅ ; D:= init. tableau des distances minimales; /*O(n)*/
2  tant_que F ≠ ∅ faire
3        choisir k ∈ F avec d_k minimal /*explorer D en O(n)*/
4        F:=F \ {k}; E:=E ∪ {k} : /*MAJ en O(1) si E,F tableaux*/
5        pour tout voisin r de k non rouge faire
6              si r ∉ F : /* cas r bleu */
7                  F:=F ∪ {r} /*O(1)*/
8              si d_k + w(k → r) < d_r :
9                    d_r:=d_k + w(k → r) /*maj tab. des distances O(1)*/
```

- Au plus $n$ transferts de $F$ vers $E$. Pour chacun recherche (ligne $L3$) du plus petit élément vert dans le tableau des distances : $O(n)$

# Complexité pour une source $e$

## Force brute pour un graphe de $n$ sommets et $p$ arcs

```
1  F:={e} ; E:=∅ ; D:= init. tableau des distances minimales; /*O(n)*/
2  tant_que F ≠ ∅ faire
3        choisir k ∈ F avec d_k minimal /*explorer D en O(n)*/
4        F:=F \ {k}; E:=E ∪ {k} : /*MAJ en O(1) si E,F tableaux*/
5        pour tout voisin r de k non rouge faire
6              si r ∉ F : /* cas r bleu */
7                 F:=F ∪ {r} /*O(1)*/
8              si d_k + w(k → r) < d_r :
9                    d_r:=d_k + w(k → r) /*maj tab. des distances O(1)*/
```

- Au plus $n$ transferts de $F$ vers $E$. Pour chacun recherche (ligne $L3$) du plus petit élément vert dans le tableau des distances : $O(n)$
- ($L5$ à $L9$) Coût des vérifications et mises à jour pour un sommet $k$ : $O(\deg^+ k)$. Au total, complexité en multiple de :

$$\sum_{k=0}^{n-1} n + \deg^+ k = n^2 + p = O(n^2)$$

# File de priorité (rappel)

- Une *file de priorité* (ou *tas-min*) de $n$ clés est une structure organisée par priorités décroissantes dans laquelle

# File de priorité (rappel)

- Une *file de priorité* (ou *tas-min*) de $n$ clés est une structure organisée par priorités décroissantes dans laquelle
  - la recherche de l'élément de priorité maximale est en $O(1)$

# File de priorité (rappel)

- Une *file de priorité* (ou *tas-min*) de $n$ clés est une structure organisée par priorités décroissantes dans laquelle
  - la recherche de l'élément de priorité maximale est en $O(1)$
  - La suppression de cet élément suivi d'une maj est en $O(\ln n)$

# File de priorité (rappel)

- Une *file de priorité* (ou *tas-min*) de $n$ clés est une structure organisée par priorités décroissantes dans laquelle
  - la recherche de l'élément de priorité maximale est en $O(1)$
  - La suppression de cet élément suivi d'une maj est en $O(\ln n)$
  - Le changement de la priorité d'un élément donné puis la maj de la file est en $O(\ln n)$

# File de priorité (rappel)

- Une *file de priorité* (ou *tas-min*) de $n$ clés est une structure organisée par priorités décroissantes dans laquelle
  - la recherche de l'élément de priorité maximale est en $O(1)$
  - La suppression de cet élément suivi d'une maj est en $O(\ln n)$
  - Le changement de la priorité d'un élément donné puis la maj de la file est en $O(\ln n)$
- Dans le cas de Dijkstra, la priorité est inversement proportionnelle à la distance à la source. Plus la distance est petite plus la priorité est grande.

# Complexité pour une source $e$

## Avec file de priorité pour un graphe de $n$ sommets et $p$ arcs

- Puisqu'on gère une structure $F$ des sommets verts et un tableau $T$ des distances à la source, on peut les fusionner en une *file de priorité* d'éléments $(d(e,s),s)$ organisée par distance croissante.

# Complexité pour une source $e$

## Avec file de priorité pour un graphe de $n$ sommets et $p$ arcs

- Puisqu'on gère une structure $F$ des sommets verts et un tableau $T$ des distances à la source, on peut les fusionner en une *file de priorité* d'éléments $(d(e,s),s)$ organisée par distance croissante.
- On implante les files de priorité comme des tas. On considère donc un tas-min (fils plus grands que père). Création par descente en $O(n)$ (toutes les distances infinies sauf $d(e,e)$). On gère en interne un tableau des positions dans le tas pour accéder en $O(1)$ à un sommet dans la file.

```
1  E:=∅;
2  T:= tas-min des couples (sommet, distance depuis e) ; /*O(n)*/
3  tant que T ≠ ∅ faire
4        retirer la racine (d(e,k),k) du tas T /*O(ln n)*/
5        si d(e,k) < +∞ alors E:=E∪{k}
6        pour tout voisin r de k non traité faire /*deg⁺ k passages*
7              si d(e,k)+w(k→r) < d(e,r): /*O(1)*/
8                    MAJ T avec d(e,r):=d(e,k)+w(k→r) /*O(ln(n))*/
```

# Complexité pour une source $e$

## Avec file de priorité pour un graphe de $n$ sommets et $p$ arcs

```
1  E:=∅;
2  T:= tas-min des couples (sommet, distance à e) ; /*O(n)*/
3  tant que T ≠ ∅ faire
4        retirer le sommet (k, d(e,k)) du tas T /*O(ln n)*/
5        si d(e,k) < +∞ alors E:=E ∪ {k}
6        pour tout voisin r de k non traité faire /*test en O(1) possibl
7              si d(e,k) + w(k,r) < d(e,r): /*accès en O(1)*/
8                    MAJ T avec d(e,r):=d(e,k) + w(k,r) /*O(ln n)*/
```

- Complexité de chaque accès/maj dans la file majoré en $O(\ln n)$. La file de priorité est de taille au plus $n$.

# Complexité pour une source $e$

## Avec file de priorité pour un graphe de $n$ sommets et $p$ arcs

```
1  E:=∅;
2  T:= tas-min des couples (sommet, distance àe) ;/*O(n)*/
3  tant que T ≠ ∅ faire
4        retirer le sommet (k,d(e,k)) du tas T /*O(ln n)*/
5        si d(e,k) < +∞ alors E:=E ∪ {k}
6        pour tout voisin r de k non traité faire /*test en O(1) possibl
7              si d(e,k) + w(k,r) < d(e,r): /*accès en O(1)*/
8                    MAJ T avec d(e,r):=d(e,k) + w(k,r) /*O(ln n)*/
```

- Complexité de chaque accès/maj dans la file majoré en $O(\ln n)$. La file de priorité est de taille au plus $n$.
- Par passage dans boucle `while` : choix puis suppression du sommet $k$ le plus prioritaire : $O(\ln n)$. Pour ses $\deg^+ k$ voisins, au plus $\deg^+ k$ maj de clés. Donc coût pour $k$ en $O((1+\deg^+ k)\ln n)$. Coût total

$$n + \sum_{e=0}^{n-1}(\deg^+ k + 1)\ln n = n + n\ln n + \ln n \sum_{e=0}^{n-1} \deg^+ k \leq n\ln n + p\ln n$$

# Complexité pour une source $e$

## Avec file de priorité pour un graphe de $n$ sommets et $p$ arcs

- Complexité en $O((n+p)\ln n)$

# Complexité pour une source $e$

## Avec file de priorité pour un graphe de $n$ sommets et $p$ arcs

- Complexité en $O((n+p)\ln n)$
- Si $p \ln n = O(n^2)$, c'est à dire si $p = O(\frac{n^2}{\ln n})$ la complexité avec file de priorités est au moins aussi bonne qu'en force brute.

# Complexité pour une source $e$

## Avec file de priorité pour un graphe de $n$ sommets et $p$ arcs

- Complexité en $O((n+p)\ln n)$
- Si $p \ln n = O(n^2)$, c'est à dire si $p = O(\frac{n^2}{\ln n})$ la complexité avec file de priorités est au moins aussi bonne qu'en force brute.
- Si le graphe est creux (peu d'arcs), $p = O(n)$ et donc la complexité avec file de priorité est $O(n \ln n)$.

# Complexité pour une source $e$

## Avec file de priorité pour un graphe de $n$ sommets et $p$ arcs

- Complexité en $O((n+p)\ln n)$
- Si $p \ln n = O(n^2)$, c'est à dire si $p = O(\frac{n^2}{\ln n})$ la complexité avec file de priorités est au moins aussi bonne qu'en force brute.
- Si le graphe est creux (peu d'arcs), $p = O(n)$ et donc la complexité avec file de priorité est $O(n \ln n)$.
- Mais si le graphe est, par exemple, complet il y a un nombre d'arcs en $p = O(n^2)$ et donc la complexité est en $O(n^2 \ln n)$.
  Dans ce cas, l'implémentation par file de priorité n'est pas intéressante.

# Variante

- Au cours de l'algorithme présenté ici, on met à jour régulièrement les priorités des voisins non rouge du sommet retiré : il faut donc, pour être efficace, maintenir un tableau des positions dans la file.
  C'est délicat.

# Variante

- Au cours de l'algorithme présenté ici, on met à jour régulièrement les priorités des voisins non rouge du sommet retiré : il faut donc, pour être efficace, maintenir un tableau des positions dans la file.
  C'est délicat.
- Il existe une variante sans mise à jour.

# Variante

- Au cours de l'algorithme présenté ici, on met à jour régulièrement les priorités des voisins non rouge du sommet retiré : il faut donc, pour être efficace, maintenir un tableau des positions dans la file.
  C'est délicat.
- Il existe une variante sans mise à jour.
  - Dans cette version, on ne met plus à jour les priorités : on se contente, pour tout voisin $v$ du sommet nouvellement retiré, d'ajouter à la file $v$ avec sa nouvelle priorité.

# Variante

- Au cours de l'algorithme présenté ici, on met à jour régulièrement les priorités des voisins non rouge du sommet retiré : il faut donc, pour être efficace, maintenir un tableau des positions dans la file.
  C'est délicat.
- Il existe une variante sans mise à jour.
  - Dans cette version, on ne met plus à jour les priorités : on se contente, pour tout voisin $v$ du sommet nouvellement retiré, d'ajouter à la file $v$ avec sa nouvelle priorité.
  - Ainsi, un même sommet peut se trouver simultannément à plusieurs endroits de la file à un moment donné.

# Présentation

On veut un PCC entre une source et une cible.

- L'algorithme de Dijkstra fonctionne bien pour trouver le chemin le plus court entre la source et tous les autres sommets (pas seulement le but).

# Présentation

On veut un PCC entre une source et une cible.

- L'algorithme de Dijkstra fonctionne bien pour trouver le chemin le plus court entre la source et tous les autres sommets (pas seulement le but).
- Il faudrait introduire une notion de « de direction vers le but ». C'est une fonction appelée *heuristique* qui permet l'inclusion d'informations supplémentaires :

# Présentation

On veut un PCC entre une source et une cible.

- L'algorithme de Dijkstra fonctionne bien pour trouver le chemin le plus court entre la source et tous les autres sommets (pas seulement le but).
- Il faudrait introduire une notion de « de direction vers le but ». C'est une fonction appelée *heuristique* qui permet l'inclusion d'informations supplémentaires :
  - L'algorithme de Dijkstra utilise la distance depuis le sommet source.

# Présentation

On veut un PCC entre une source et une cible.

- L'algorithme de Dijkstra fonctionne bien pour trouver le chemin le plus court entre la source et tous les autres sommets (pas seulement le but).
- Il faudrait introduire une notion de « de direction vers le but ». C'est une fonction appelée *heuristique* qui permet l'inclusion d'informations supplémentaires :
  - L'algorithme de Dijkstra utilise la distance depuis le sommet source.
  - L'algorithme A* utilise à la fois la distance depuis le sommet source et la distance estimée jusqu'au but (l'heuristique).

# Présentation

On veut un PCC entre une source et une cible.

- L'algorithme de Dijkstra fonctionne bien pour trouver le chemin le plus court entre la source et tous les autres sommets (pas seulement le but).
- Il faudrait introduire une notion de « de direction vers le but ». C'est une fonction appelée *heuristique* qui permet l'inclusion d'informations supplémentaires :
  - L'algorithme de Dijkstra utilise la distance depuis le sommet source.
  - L'algorithme A* utilise à la fois la distance depuis le sommet source et la distance estimée jusqu'au but (l'heuristique).
- L'algorithme A* est donc une variante de Dijkstra où on cherche à éviter les explorations inutiles en introduisant une heuristique.

# Construction

- A* sélectionne le chemin qui, pour un sommet $n$, minimise la fonction $f$ suivante dite *fonction de score* :

$$f(n) = d(n) + h(n)$$

# Construction

- A* sélectionne le chemin qui, pour un sommet $n$, minimise la fonction $f$ suivante dite *fonction de score* :

$$f(n) = d(n) + h(n)$$

  - $d(n)$ est le coût du chemin du point de départ au sommet $n$

# Construction

- A* sélectionne le chemin qui, pour un sommet $n$, minimise la fonction $f$ suivante dite *fonction de score* :

$$f(n) = d(n) + h(n)$$

  - $d(n)$ est le coût du chemin du point de départ au sommet $n$
  - $h(n)$ (l'heuristique) est le coût estimé du chemin du sommet $n$ au sommet de destination (exemple : *distance de Manhattan*). Par hypothèse : $h(\text{but}) = 0$ et $h(u) \geq 0$ pour tout sommet $u$.

# Construction

- A* sélectionne le chemin qui, pour un sommet $n$, minimise la fonction $f$ suivante dite *fonction de score* :

$$f(n) = d(n) + h(n)$$

  - $d(n)$ est le coût du chemin du point de départ au sommet $n$
  - $h(n)$ (l'heuristique) est le coût estimé du chemin du sommet $n$ au sommet de destination (exemple : *distance de Manhattan*). Par hypothèse : $h(\text{but}) = 0$ et $h(u) \geq 0$ pour tout sommet $u$.
- Ainsi, $d(n) + h(n)$ est le coût estimé d'un chemin de la source au but s'il passe par $n$.

# Construction

- A* sélectionne le chemin qui, pour un sommet $n$, minimise la fonction $f$ suivante dite *fonction de score* :

$$f(n) = d(n) + h(n)$$

  - $d(n)$ est le coût du chemin du point de départ au sommet $n$
  - $h(n)$ (l'heuristique) est le coût estimé du chemin du sommet $n$ au sommet de destination (exemple : *distance de Manhattan*). Par hypothèse : $h(\text{but}) = 0$ et $h(u) \geq 0$ pour tout sommet $u$.
- Ainsi, $d(n) + h(n)$ est le coût estimé d'un chemin de la source au but s'il passe par $n$.
- Si $h$ est la fonction identiquement nulle, alors on retombe sur l'algorithme de Dijkstra.

# Algorithme admissible

- Un algo de recherche de PCC qui garantit de toujours trouver le PCC d'une source à un but est dit *admissible* (ex : Dijkstra)

# Algorithme admissible

- Un algo de recherche de PCC qui garantit de toujours trouver le PCC d'une source à un but est dit *admissible* (ex : Dijkstra)
- Si A* utilise une heuristique *admissible* (c.a.d qui ne surestime jamais le coût réel pour joindre le but), A* est lui même admissible.

# Algorithme admissible

- Un algo de recherche de PCC qui garantit de toujours trouver le PCC d'une source à un but est dit *admissible* (ex : Dijkstra)
- Si A* utilise une heuristique *admissible* (c.a.d qui ne surestime jamais le coût réel pour joindre le but), A* est lui même admissible.
- L'heuristique à utiliser dépend du problème auquel on veut appliquer A*.
  Exemples d'heuristiques :

# Algorithme admissible

- Un algo de recherche de PCC qui garantit de toujours trouver le PCC d'une source à un but est dit *admissible* (ex : Dijkstra)
- Si A* utilise une heuristique *admissible* (c.a.d qui ne surestime jamais le coût réel pour joindre le but), A* est lui même admissible.
- L'heuristique à utiliser dépend du problème auquel on veut appliquer A*.
  Exemples d'heuristiques :
  - heuristique nulle : on retrouve Dijkstra

# Algorithme admissible

- Un algo de recherche de PCC qui garantit de toujours trouver le PCC d'une source à un but est dit *admissible* (ex : Dijkstra)
- Si A* utilise une heuristique *admissible* (c.a.d qui ne surestime jamais le coût réel pour joindre le but), A* est lui même admissible.
- L'heuristique à utiliser dépend du problème auquel on veut appliquer A*.
  Exemples d'heuristiques :
  - heuristique nulle : on retrouve Dijkstra
  - distance à vol d'oiseau (pour des points sur une carte)

# Algorithme admissible

- Un algo de recherche de PCC qui garantit de toujours trouver le PCC d'une source à un but est dit *admissible* (ex : Dijkstra)
- Si A* utilise une heuristique *admissible* (c.a.d qui ne surestime jamais le coût réel pour joindre le but), A* est lui même admissible.
- L'heuristique à utiliser dépend du problème auquel on veut appliquer A*.
  Exemples d'heuristiques :
  - heuristique nulle : on retrouve Dijkstra
  - distance à vol d'oiseau (pour des points sur une carte)
  - *distance de Manhattan* pour des cases dans un labyrinthe.

# Algorithme admissible

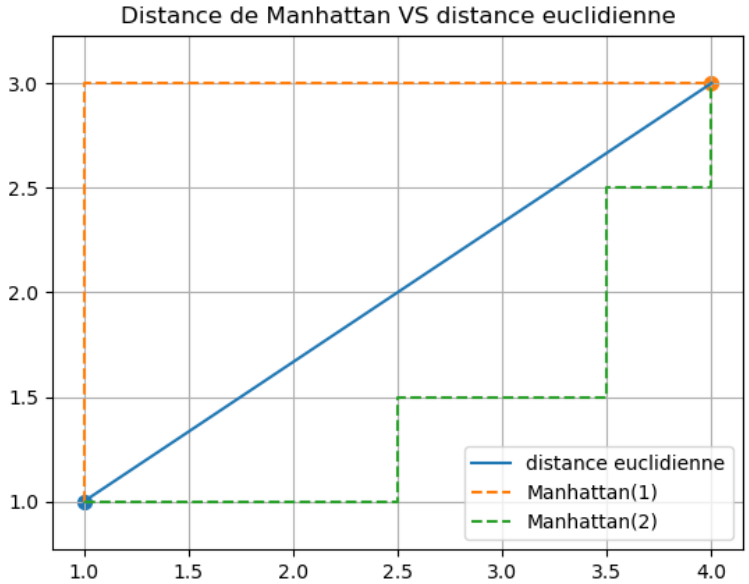
- Un algo de recherche de PCC qui garantit de toujours trouver le PCC d'une source à un but est dit *admissible* (ex : Dijkstra)
- Si A* utilise une heuristique *admissible* (c.a.d qui ne surestime jamais le coût réel pour joindre le but), A* est lui même admissible.
- L'heuristique à utiliser dépend du problème auquel on veut appliquer A*.
  Exemples d'heuristiques :
  - heuristique nulle : on retrouve Dijkstra
  - distance à vol d'oiseau (pour des points sur une carte)
  - *distance de Manhattan* pour des cases dans un labyrinthe.
- La meilleure heuristique possible est celle qui donne la distance minimale réelle (par ex : distance à vol d'oiseau sur une carte) mais elle est souvent impraticable.

# Distance de Manhattan

- La *distance de Manhattan* est une distance fréquemment utilisée pour les environnement basés sur des grilles (comme les labyrinthes) où les mouvements sont uniquement verticaux ou horizontaux.

# Distance de Manhattan

- La *distance de Manhattan* est une distance fréquemment utilisée pour les environnement basés sur des grilles (comme les labyrinthes) où les mouvements sont uniquement verticaux ou horizontaux.
- La distance de Manhattan est la simple somme des mouvements horizontaux et verticaux, alors que la distance euclidienne est la norme du vecteur qui joint deux points de la grille.



# Algorithme

$w(u, v)$ désigne le poids de l'arc $(u, v)$ (-1 si cet arc n'existe pas).

```
1  fonction astar(G: graphe,s : source,b : but, h : fct heuristique)
2      /*les sommets en cours de traitement*/
3      F:=file vide;
4      p:=tableau des predécesseurs; /*initialisé à -1*/
5      d:=table des distances;/*comme dans Dijkstra*/
6      f:=table des scores; /*f[n]=d[n]+h(n); initialisé à -1*/
7      d[s]:=0; p[s]:=s; f[s]:=h(s);
8      ajouter s à F;
9      tant_que F≠∅ faire
10         u:=retirer le sommet de F avec plus petit score
11         si u=b alors renvoyer d,p;/*on a atteint le but*/
12         pour tout voisin v de u faire
13           tentative_distance:=d[u]+w(u,v);
14           si tentative_distance<d[v] alors
15              p[v]:=u; d[v]:=tentative_distance;
16              f[v]:=tentative_distance+h(v);
17              si v∉F alors ajouter v àF;/*v peut revenir dans F*/
18         fin_faire
19     fin_faire
20     soulever une exception NOT_FOUND
```

# Différences avec Dijkstra

Observons les différences avec Dijkstra :

- Le sommet qu'on retire de la file n'est pas celui avec la plus petite distance (Dijkstra) à la source mais le plus petit score (A*).

# Différences avec Dijkstra

Observons les différences avec Dijkstra :

- Le sommet qu'on retire de la file n'est pas celui avec la plus petite distance (Dijkstra) à la source mais le plus petit score (A*).
- Dans Dijkstra, on examine les voisins du sommet courant qui ne sont jamais sortis de la file.
  Dans A* on examine tous les voisins y compris un voisin qui est déjà sorti de la file : Un même sommet peut donc sortir et revenir plusieurs fois dans la file.

# Différences avec Dijkstra

Observons les différences avec Dijkstra :

- Le sommet qu'on retire de la file n'est pas celui avec la plus petite distance (Dijkstra) à la source mais le plus petit score (A*).
- Dans Dijkstra, on examine les voisins du sommet courant qui ne sont jamais sortis de la file.
  Dans A* on examine tous les voisins y compris un voisin qui est déjà sorti de la file : Un même sommet peut donc sortir et revenir plusieurs fois dans la file.
- C'est tout.

# Différences avec Dijkstra

Observons les différences avec Dijkstra :

- Le sommet qu'on retire de la file n'est pas celui avec la plus petite distance (Dijkstra) à la source mais le plus petit score (A*).
- Dans Dijkstra, on examine les voisins du sommet courant qui ne sont jamais sortis de la file.
  Dans A* on examine tous les voisins y compris un voisin qui est déjà sorti de la file : Un même sommet peut donc sortir et revenir plusieurs fois dans la file.
- C'est tout.
- Mais la complexité de A* peut malheureusement être exponentielle (puisque les sommets peuvent entrer et sortir de la file plsuieurs fois) !

# Heuristique monotone

- Une heuristique $h$ est dite *monotone* quand pour tous sommets $x, y$ on a

$$h(x) \leq d(x, y) + h(y)$$

où $d(x, y)$ est le coût d'un arc de $x$ à $y$ ($+\infty$ si par d'arc).

# Heuristique monotone

- Une heuristique $h$ est dite *monotone* quand pour tous sommets $x, y$ on a

$$h(x) \leq d(x, y) + h(y)$$

où $d(x, y)$ est le coût d'un arc de $x$ à $y$ ($+\infty$ si par d'arc).

- Dans une heuristique monotone, la différence des heuristiques de deux sommets ne surestime jamais la distance réelle entre les deux sommets.

# Heuristique monotone

- Une heuristique $h$ est dite *monotone* quand pour tous sommets $x, y$ on a
$$h(x) \leq d(x, y) + h(y)$$
où $d(x, y)$ est le coût d'un arc de $x$ à $y$ ($+\infty$ si par d'arc).
- Dans une heuristique monotone, la différence des heuristiques de deux sommets ne surestime jamais la distance réelle entre les deux sommets.
- Il est rare que les heuristiques soient monotone.
Mais si une heuristique est monotone, alors on peut montrer que la complexité temporelle de l'algorithme A* est polynomiale.

# Heuristique monotone

- Une heuristique $h$ est dite *monotone* quand pour tous sommets $x, y$ on a
$$h(x) \leq d(x, y) + h(y)$$
où $d(x, y)$ est le coût d'un arc de $x$ à $y$ ($+\infty$ si par d'arc).
- Dans une heuristique monotone, la différence des heuristiques de deux sommets ne surestime jamais la distance réelle entre les deux sommets.
- Il est rare que les heuristiques soient monotone.
Mais si une heuristique est monotone, alors on peut montrer que la complexité temporelle de l'algorithme A* est polynomiale.
- En ce sens, avec une heuristique monotone l'algorithme A* se comporte comme un Dijkstra dans lequel la fonction de coût est $d'(x, y) = d(x, y) + h(y) - h(x)$ plutôt que $d(x, y)$.

# Monotonie et admissibilité

**Proposition**

*Une heuristique monotone est admissible.*

- On cherche à montrer que pour tout chemin

$$v_0 \xrightarrow{d_0} v_1 \xrightarrow{d_1} v_2 \cdots v_n \xrightarrow{d_n} b = \text{but}$$

on a $h(v_0) \leq d_0 + d_1 + \cdots + d_n$. Par récurrence sur $n$.

# Monotonie et admissibilité

## Proposition

*Une heuristique monotone est admissible.*

- On cherche à montrer que pour tout chemin

$$v_0 \xrightarrow{d_0} v_1 \xrightarrow{d_1} v_2 \cdots v_n \xrightarrow{d_n} b = \text{but}$$

  on a $h(v_0) \leq d_0 + d_1 + \cdots + d_n$. Par récurrence sur $n$.
- Si $n = 0$, on a $v_0 = b$ et $h(b) = 0$ par hypothèse. OK.

# Monotonie et admissibilité

**Proposition**

*Une heuristique monotone est admissible.*

- On cherche à montrer que pour tout chemin

$$v_0 \xrightarrow{d_0} v_1 \xrightarrow{d_1} v_2 \cdots v_n \xrightarrow{d_n} b = \text{but}$$

  on a $h(v_0) \leq d_0 + d_1 + \cdots + d_n$. Par récurrence sur $n$.
- Si $n = 0$, on a $v_0 = b$ et $h(b) = 0$ par hypothèse. OK.
- Pour $n > 0$, on a $h(v_0) \leq d_0 + h(v_1)$ car $h$ est monotone. Par HR, $h(v_1) \leq d_1 + \cdots + d_n$ et donc

$$h(v_0) \leq d_0 + h(v_1) \leq d_0 + (d_1 + \cdots + d_n)$$

  IZP.

# Monotonie et admissibilité

**Proposition**

*Une heuristique monotone est admissible.*

- On cherche à montrer que pour tout chemin

$$v_0 \xrightarrow{d_0} v_1 \xrightarrow{d_1} v_2 \cdots v_n \xrightarrow{d_n} b = \text{but}$$

  on a $h(v_0) \leq d_0 + d_1 + \cdots + d_n$. Par récurrence sur $n$.
- Si $n = 0$, on a $v_0 = b$ et $h(b) = 0$ par hypothèse. OK.
- Pour $n > 0$, on a $h(v_0) \leq d_0 + h(v_1)$ car $h$ est monotone. Par HR, $h(v_1) \leq d_1 + \cdots + d_n$ et donc

$$h(v_0) \leq d_0 + h(v_1) \leq d_0 + (d_1 + \cdots + d_n)$$

  IZP.
- Ainsi, l'heuristique monotone ne surestime jamais la distance réelle : elle est admissible.