Semestre 2 / Cours 4 - Parcours de graphes pondérés & Applications

ITC MPSI & PCSI – Année 2023-2024

Sommaire

1. Parcours de graphes pondérés

2. Algorithme A*

Objectifs

- Vocabulaire : longueur d’un chemin, distance entre deux sommets, plus court chemin entre deux sommets,

Objectifs

- Vocabulaire : longueur d'un chemin, distance entre deux sommets, plus court chemin entre deux sommets,
- savoir adapter l'algorithme de parcours en largeur pour déterminer *un* plus court chemin entre deux sommets d'un graphe *non pondéré*,

Objectifs

- Vocabulaire : longueur d'un chemin, distance entre deux sommets, plus court chemin entre deux sommets,
- savoir adapter l'algorithme de parcours en largeur pour déterminer *un* plus court chemin entre deux sommets d'un graphe *non pondéré*,
- savoir mettre en oeuvre l'algorithme de Dijkstra pour déterminer *un* plus court chemin entre deux sommets d'un graphe *pondéré*,

Objectifs

- Vocabulaire : longueur d'un chemin, distance entre deux sommets, plus court chemin entre deux sommets,
- savoir adapter l'algorithme de parcours en largeur pour déterminer *un* plus court chemin entre deux sommets d'un graphe *non pondéré*,
- savoir mettre en oeuvre l'algorithme de Dijkstra pour déterminer *un* plus court chemin entre deux sommets d'un graphe *pondéré*,
- savoir orthographier correctement « Dijkstra »,

Objectifs

- Vocabulaire : longueur d'un chemin, distance entre deux sommets, plus court chemin entre deux sommets,
- savoir adapter l'algorithme de parcours en largeur pour déterminer *un* plus court chemin entre deux sommets d'un graphe *non pondéré*,
- savoir mettre en oeuvre l'algorithme de Dijkstra pour déterminer *un* plus court chemin entre deux sommets d'un graphe *pondéré*,
- savoir orthographier correctement « Dijkstra »,
- introduire la notion d'« heurisitique » qui permet, dans la plupart des cas, d'accélérer la recherche de plus court chemin. Cette notion est mise en place avec l'algorithme A*.

Parcours de graphes pondérés

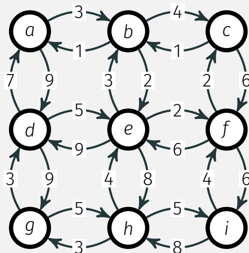


Figure 1 – Graphe G_2 orienté et pondéré.

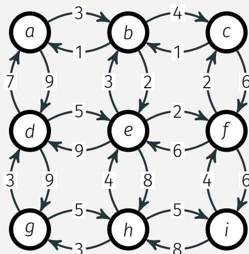


Figure 1 – Graphe G_2 orienté et pondéré.

- La *longueur* ou *poids* d'un chemin dans un graphe pondéré est désormais définie comme la somme des poids de ses arêtes (arcs s'il est orienté).

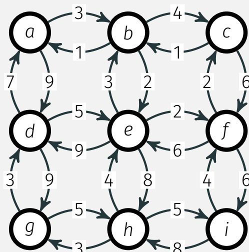


Figure 1 – Graphe G_2 orienté et pondéré.

- La *longueur* ou *poids* d'un chemin dans un graphe pondéré est désormais définie comme la somme des poids de ses arêtes (arcs s'il est orienté).
- Exemple : $\gamma = (e, d, g, h, i)$, $\delta(\gamma) = 9 + 9 + 5 + 5 = 28$

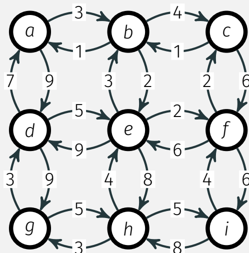


Figure 1 – Graphe G_2 orienté et pondéré.

- La définition de la *distance* entre deux sommets n'est pas modifiée : c'est la longueur *d'un* chemin de longueur minimale entre les deux sommets.

Plus court chemin

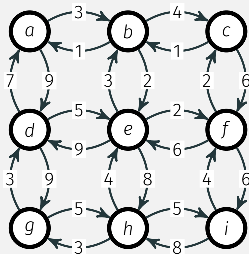


Figure 1 – Graphe G_2 orienté et pondéré.

- La définition de la *distance* entre deux sommets n'est pas modifiée : c'est la longueur *d'un* chemin de longueur minimale entre les deux sommets.
- Exemple : $d_c[f] = 5$, atteinte avec le chemin (c, b, e, f). Notons que le chemin (c, f), qui ne contient pourtant qu'un arc, est plus long : $\delta(c, f) = 6$.

- Le parcours en largeur permet de trouver des chemins avec un nombre minimum d’arêtes, mais ils ne sont pas toujours les plus courts (voir exemple précédent).

Une adaptation de BFS

- Le parcours en largeur permet de trouver des chemins avec un nombre minimum d'arêtes, mais ils ne sont pas toujours les plus courts (voir exemple précédent).
- Adaptation possible : sur chaque arête de poids $p \geq 2$, insérer $(p-1)$ sommets intermédiaires ; toutes les arêtes portant désormais un poids 1.

Une adaptation de BFS

- Le parcours en largeur permet de trouver des chemins avec un nombre minimum d'arêtes, mais ils ne sont pas toujours les plus courts (voir exemple précédent).
- Adaptation possible : sur chaque arête de poids $p \geq 2$, insérer $(p-1)$ sommets intermédiaires ; toutes les arêtes portant désormais un poids 1.
- Inconvénients :
  - ne fonctionne que pour des poids entiers,

Une adaptation de BFS

- Le parcours en largeur permet de trouver des chemins avec un nombre minimum d'arêtes, mais ils ne sont pas toujours les plus courts (voir exemple précédent).
- Adaptation possible : sur chaque arête de poids $p \geq 2$, insérer $(p - 1)$ sommets intermédiaires ; toutes les arêtes portant désormais un poids 1.
- Inconvénients :
  - ◇ ne fonctionne que pour des poids entiers,
  - ◇ augmente artificiellement le nombre de sommets et donc le temps d'exécution.

Principe de sous-optimalité, conséquence et utilisation

- *Principe de sous-optimalité* : soit c un *plus court chemin* d'un sommet u vers un sommet v d'un graphe. Notons $u \overset{c}{\rightsquigarrow} v$ un tel plus court chemin. Si c passe par un sommet intermédiaire s, alors $u \overset{c_1}{\rightsquigarrow} s$ et $s \overset{c_2}{\rightsquigarrow} v$ sont aussi des plus courts chemins.

Principe de sous-optimalité, conséquence et utilisation

- *Principe de sous-optimalité* : soit c un *plus court chemin* d'un sommet u vers un sommet v d'un graphe. Notons $u \overset{c}{\rightsquigarrow} v$ un tel plus court chemin. Si c passe par un sommet intermédiaire s, alors $u \overset{c_1}{\rightsquigarrow} s$ et $s \overset{c_2}{\rightsquigarrow} v$ sont aussi des plus courts chemins.
- *Conséquence* : déterminer un plus court chemin entre deux sommets u et v fournit des plus courts chemins entre u et tous les sommets situés sur le chemin aboutissant en v.

Principe de sous-optimalité, conséquence et utilisation

- *Principe de sous-optimalité* : soit c un *plus court chemin* d'un sommet u vers un sommet v d'un graphe. Notons $u \overset{c}{\rightsquigarrow} v$ un tel plus court chemin. Si c passe par un sommet intermédiaire s, alors $u \overset{c_1}{\rightsquigarrow} s$ et $s \overset{c_2}{\rightsquigarrow} v$ sont aussi des plus courts chemins.
- *Conséquence* : déterminer un plus court chemin entre deux sommets u et v fournit des plus courts chemins entre u et tous les sommets situés sur le chemin aboutissant en v.
- *Principe de l'algorithme de* DIJKSTRA : utiliser le principe de sous-optimalité pour déterminer, de proche en proche, les distances d'un sommet de départ u vers chacun des sommets du graphe.

Principe de sous-optimalité, conséquence et utilisation

- *Principe de sous-optimalité* : soit c un *plus court chemin* d'un sommet u vers un sommet v d'un graphe. Notons $u \overset{c}{\rightsquigarrow} v$ un tel plus court chemin. Si c passe par un sommet intermédiaire s, alors $u \overset{c_1}{\rightsquigarrow} s$ et $s \overset{c_2}{\rightsquigarrow} v$ sont aussi des plus courts chemins.
- *Conséquence* : déterminer un plus court chemin entre deux sommets u et v fournit des plus courts chemins entre u et tous les sommets situés sur le chemin aboutissant en v.
- *Principe de l'algorithme de* DIJKSTRA : utiliser le principe de sous-optimalité pour déterminer, de proche en proche, les distances d'un sommet de départ u vers chacun des sommets du graphe.En pratique, un *dictionnaire* des distances d_u est initialisé avec des clés égales aux étiquettes des sommets et des valeurs égales à l'infini, excepté pour le sommet de départ pour lequel la distance est zéro.

Principe de sous-optimalité, conséquence et utilisation

- *Principe de sous-optimalité* : soit c un *plus court chemin* d'un sommet u vers un sommet v d'un graphe. Notons $u \overset{c}{\rightsquigarrow} v$ un tel plus court chemin. Si c passe par un sommet intermédiaire s, alors $u \overset{c_1}{\rightsquigarrow} s$ et $s \overset{c_2}{\rightsquigarrow} v$ sont aussi des plus courts chemins.
- *Conséquence* : déterminer un plus court chemin entre deux sommets u et v fournit des plus courts chemins entre u et tous les sommets situés sur le chemin aboutissant en v.
- *Principe de l'algorithme de Dijkstra* : utiliser le principe de sous-optimalité pour déterminer, de proche en proche, les distances d'un sommet de départ u vers chacun des sommets du graphe.En pratique, un *dictionnaire* des distances d_u est initialisé avec des clés égales aux étiquettes des sommets et des valeurs égales à l'infini, excepté pour le sommet de départ pour lequel la distance est zéro. En Python, il est possible de définir un tel infini par `float("inf")`, cet objet étant reconnu par Python comme l'équivalent de l'infini, que ce soit pour des entiers ou des flottants !

Algorithme de Dijkstra : initialisation et première étape

Dans la présentation de l'algorithme qui suit, nous reprenons le vocabulaire adopté dans le précédent chapitre : un sommet est soit *non découvert*, soit *découvert*, soit *visité*.

Algorithme de Dijkstra : initialisation et première étape

Dans la présentation de l'algorithme qui suit, nous reprenons le vocabulaire adopté dans le précédent chapitre : un sommet est soit *non découvert*, soit *découvert*, soit *visité*.

- *Initialisation :*
  Si $G = (S, A)$ est un graphe pondéré par des poids positifs définis par la fonction de pondération p, l'*initialisation* de l'algorithme consiste en :

$$\mathrm{d}_u[u] = 0, \quad \text{et :} \quad \forall v \in S \backslash \{u\}, \quad \mathrm{d}_u[v] = +\infty.$$

Algorithme de Dijkstra : initialisation et première étape

Dans la présentation de l'algorithme qui suit, nous reprenons le vocabulaire adopté dans le précédent chapitre : un sommet est soit *non découvert*, soit *découvert*, soit *visité*.

- *Initialisation :*
  Si $G = (S, A)$ est un graphe pondéré par des poids positifs définis par la fonction de pondération p, l'*initialisation* de l'algorithme consiste en :

$$\mathrm{d}_u[u] = 0, \quad \text{et :} \quad \forall v \in S \backslash \{u\}, \quad \mathrm{d}_u[v] = +\infty.$$

- *Étape 1 :*
  u est marqué comme *visité* et les sommets *voisins non visités* de u, notés $v_{1,1}, \ldots, v_{1,n_1}$, sont *découverts* puis on « *relâche* » chaque arc $(u, v_{1,i})$:

Algorithme de Dijkstra : initialisation et première étape

Dans la présentation de l'algorithme qui suit, nous reprenons le vocabulaire adopté dans le précédent chapitre : un sommet est soit *non découvert*, soit *découvert*, soit *visité*.

- *Initialisation :*
  Si $G = (S, A)$ est un graphe pondéré par des poids positifs définis par la fonction de pondération p, l'*initialisation* de l'algorithme consiste en :

$$\mathrm{d}_u[u] = 0, \quad \text{et :} \quad \forall v \in S \backslash \{u\}, \quad \mathrm{d}_u[v] = +\infty.$$

- *Étape 1 :*
  u est marqué comme *visité* et les sommets *voisins non visités* de u, notés $v_{1,1}, \ldots, v_{1,n_1}$, sont *découverts* puis on « *relâche* » chaque arc $(u, v_{1,i})$:
  - si $d_u[u] + p(u, v_{1,i}) < d_u[v_{1,i}]$ alors $d_u[v_{1,i}]$ est redéfinie par $d_u[u] + p(u, v_{1,i})$;

Algorithme de Dijkstra : initialisation et première étape

Dans la présentation de l'algorithme qui suit, nous reprenons le vocabulaire adopté dans le précédent chapitre : un sommet est soit *non découvert*, soit *découvert*, soit *visité*.

- *Initialisation :*
  Si $G = (S, A)$ est un graphe pondéré par des poids positifs définis par la fonction de pondération p, l'*initialisation* de l'algorithme consiste en :

$$\mathrm{d}_u[u] = 0, \quad \text{et :} \quad \forall v \in S \backslash \{u\}, \quad \mathrm{d}_u[v] = +\infty.$$

- *Étape 1 :*
  u est marqué comme *visité* et les sommets *voisins non visités* de u, notés $v_{1,1}, \ldots, v_{1,n_1}$, sont *découverts* puis on « *relâche* » chaque arc $(u, v_{1,i})$:
  - si $d_u[u] + p(u, v_{1,i}) < d_u[v_{1,i}]$ alors $d_u[v_{1,i}]$ est redéfinie par $d_u[u] + p(u, v_{1,i})$;
  - sinon la valeur de $d_u[v_{1,i}]$ reste inchangée.

Algorithme de Dijkstra : initialisation et première étape

Dans la présentation de l'algorithme qui suit, nous reprenons le vocabulaire adopté dans le précédent chapitre : un sommet est soit *non découvert*, soit *découvert*, soit *visité*.

- *Initialisation :*
  Si $G = (S, A)$ est un graphe pondéré par des poids positifs définis par la fonction de pondération p, l'*initialisation* de l'algorithme consiste en :

$$\mathrm{d}_u[u] = 0, \quad \text{et :} \quad \forall v \in S \backslash \{u\}, \quad \mathrm{d}_u[v] = +\infty.$$

- *Étape 1 :*
  u est marqué comme *visité* et les sommets *voisins non visités* de u, notés $v_{1,1}, \ldots, v_{1,n_1}$, sont *découverts* puis on « *relâche* » chaque arc $(u, v_{1,i})$:
  - si $d_u[u] + p(u, v_{1,i}) < d_u[v_{1,i}]$ alors $d_u[v_{1,i}]$ est redéfinie par $d_u[u] + p(u, v_{1,i})$;
  - sinon la valeur de $d_u[v_{1,i}]$ reste inchangée.

Algorithme de Dijkstra : initialisation et première étape

À l'issue de cette première étape, toutes les distances $\mathrm{d}_u[v_{1,i}]$ étant initialement infinies, le dictionnaire d_u devient :

$$\mathrm{d}_u[u] = 0, \quad \text{et :} \quad \begin{cases} \forall i \in [\![1, n_1]\!], & \mathrm{d}_u[v_{1,i}] & = p(u, v_{1,i}) \\ \forall v \in S \backslash \{u, v_1, \dots, v_{1,n_1}\}, & \mathrm{d}_u[v] & = +\infty. \end{cases}$$

Algorithme de Dijkstra : étapes suivantes

- *Étape 2.*
  Parmi tous les sommets découverts et non visités, l'algorithme identifie ensuite le sommet, noté v_1, de distance **minimale** $d_u[v_1]$. Le principe de sous-optimalité assure alors que cette distance est la plus courte distance entre u et v_1. Sa valeur dans d_u ne doit plus évoluer.

Algorithme de Dijkstra : étapes suivantes

- *Étape 2.*
  Parmi tous les sommets découverts et non visités, l'algorithme identifie ensuite le sommet, noté v_1, de distance **minimale** $d_u[v_1]$. Le principe de sous-optimalité assure alors que cette distance est la plus courte distance entre u et v_1. Sa valeur dans d_u ne doit plus évoluer.
  Le sommet v_1 sert alors de nouveau somme de départ et est marqué comme *visité*. Les n_2 sommets voisins *non visités* de v_1, notés $v_{2,1}, \ldots, v_{2,n_2}$, sont alors *découverts* et on *relâche* chaque arc $(v_1, v_{2,i})$

Algorithme de Dijkstra : étapes suivantes

- *Étape 2.*
  Parmi tous les sommets découverts et non visités, l'algorithme identifie ensuite le sommet, noté v_1, de distance **minimale** $d_u[v_1]$. Le principe de sous-optimalité assure alors que cette distance est la plus courte distance entre u et v_1. Sa valeur dans d_u ne doit plus évoluer.
  Le sommet v_1 sert alors de nouveau somme de départ et est marqué comme *visité*. Les n_2 sommets voisins *non visités* de v_1, notés $v_{2,1}, \ldots, v_{2,n_2}$, sont alors *découverts* et on *relâche* chaque arc $(v_1, v_{2,i})$
  - Si $d_u[v_1] + p(v_1, v_{2,i}) < d_u[v_{2,i}]$ alors $d_u[v_{2,i}]$ est redéfinie par $d_u[v_1] + p(v_1, v_{2,i})$.

Algorithme de Dijkstra : étapes suivantes

- *Étape 2.*
  Parmi tous les sommets découverts et non visités, l'algorithme identifie ensuite le sommet, noté v_1, de distance **minimale** $d_u[v_1]$. Le principe de sous-optimalité assure alors que cette distance est la plus courte distance entre u et v_1. Sa valeur dans d_u ne doit plus évoluer.
  Le sommet v_1 sert alors de nouveau somme de départ et est marqué comme *visité*. Les n_2 sommets voisins *non visités* de v_1, notés $v_{2,1}, \ldots, v_{2,n_2}$, sont alors *découverts* et on *relâche* chaque arc $(v_1, v_{2,i})$
  - Si $d_u[v_1] + p(v_1, v_{2,i}) < d_u[v_{2,i}]$ alors $d_u[v_{2,i}]$ est redéfinie par $d_u[v_1] + p(v_1, v_{2,i})$.
  - Sinon la valeur de $d_u[v_{2,i}]$ reste inchangée.

Algorithme de Dijkstra : étapes suivantes

- *Étape 2.*
  Parmi tous les sommets découverts et non visités, l'algorithme identifie ensuite le sommet, noté v_1, de distance **minimale** $d_u[v_1]$. Le principe de sous-optimalité assure alors que cette distance est la plus courte distance entre u et v_1. Sa valeur dans d_u ne doit plus évoluer.
  Le sommet v_1 sert alors de nouveau somme de départ et est marqué comme *visité*. Les n_2 sommets voisins *non visités* de v_1, notés $v_{2,1}, \ldots, v_{2,n_2}$, sont alors *découverts* et on *relâche* chaque arc $(v_1, v_{2,i})$
  - ◇ Si $d_u[v_1] + p(v_1, v_{2,i}) < d_u[v_{2,i}]$ alors $d_u[v_{2,i}]$ est redéfinie par $d_u[v_1] + p(v_1, v_{2,i})$.
  - ◇ Sinon la valeur de $d_u[v_{2,i}]$ reste inchangée.
- Ces étapes sont répétées tant que l'ensemble des sommets *découverts et non visités* n'est pas vide. Quand cet ensemble est vide, tous les sommets ont été visités et les valeurs alors contenues dans d_u sont les plus courtes distances de u à chacun des sommets du graphe.

Files de priorité

- Une file de priorité est une structure de donnée dynamique dans laquelle les données sont réorganisées à chaque ajout/suppression. Son intérêt est de permettre l'extraction à moindre coût un élément de *priorité* minimale.

Files de priorité

- Une file de priorité est une structure de donnée dynamique dans laquelle les données sont réorganisées à chaque ajout/suppression. Son intérêt est de permettre l'extraction à moindre coût un élément de *priorité* **minimale**.
- Nous utiliserons un module spécifique qui implémente une telle structure : le module `heapq` préalablement chargé par l'instruction

```
import heapq
```

Files de priorité

- Manipulation :
  - ◇ **Création d'une file de priorité.** L'instruction `heapq.heapify(hq)` transforme la liste `hq` en une file de priorité ; la complexité de l'opération est en $O(n)$.
  - ◇ **Longueur d'une priorité file de priorité.** L'instruction `len(hq)` renvoie le nombre d'éléments de a file de priorité `hq` ; la complexité est en $O(1)$.
  - ◇ **Ajout.** L'instruction `heapq.heappush(hq,x)` ajoute un élément `x` dans la file de priorité `hq` puis réorganise les données ; la complexité est en $O(\log n)$.

Files de priorité

- Manipulation :
  - **Création d'une file de priorité.** L'instruction `heapq.heapify(hq)` transforme la liste **hq** en une file de priorité ; la complexité de l'opération est en $O(n)$.
  - **Longueur d'une priorité file de priorité.** L'instruction `len(hq)` renvoie le nombre d'éléments de a file de priorité **hq** ; la complexité est en $O(1)$.
  - **Ajout.** L'instruction `heapq.heappush(hq,x)` ajoute un élément **x** dans la file de priorité **hq** puis réorganise les données ; la complexité est en $O(\log n)$.
  - **Extraction.** L'instruction `heapq.heappop(hq)` renvoie l'élément de *priorité minimale* de la file de priorité **hq**, puis le supprime de **hq** et réorganise les données ; la complexité est en $O(\log n)$.

Files de priorité

- Manipulation :
  - **Création d'une file de priorité.** L'instruction `heapq.heapify(hq)` transforme la liste `hq` en une file de priorité ; la complexité de l'opération est en $O(n)$.
  - **Longueur d'une priorité file de priorité.** L'instruction `len(hq)` renvoie le nombre d'éléments de a file de priorité `hq` ; la complexité est en $O(1)$.
  - **Ajout.** L'instruction `heapq.heappush(hq,x)` ajoute un élément `x` dans la file de priorité `hq` puis réorganise les données ; la complexité est en $O(\log n)$.
  - **Extraction.** L'instruction `heapq.heappop(hq)` renvoie l'élément de *priorité minimale* de la file de priorité `hq`, puis le supprime de `hq` et réorganise les données ; la complexité est en $O(\log n)$.
    Si la file est vide, un message d'erreur est renvoyé.

Exemple

```
>>> import heapq
>>> import numpy as np
>>> a, b = 25, 75
>>> n = 10
>>> hq = [np.random.randint(a,b) for _ in range(n)] \
↪ # Liste de n entiers tirés au hasard
>>> hq
[31, 55, 31, 31, 46, 46, 39, 48, 74, 45]
>>> heapq.heapify(hq) # Définition d'une file de \
↪ priorité hq
>>> hq
[31, 31, 31, 48, 45, 46, 39, 55, 74, 46]
```

Exemple

```
>>> while len(hq) > 0: # Extraction des entiers de hq
...     x = heapq.heappop(hq)
...     print(x)
...
31
31
31
39
45
46
46
48
55
74
```

Exemple avec poids

```
>>> hq = [(10, 'a'), (5, 'b'), (2, 'c'), (8, 'd')] # \
↪ liste de couples
>>> hq
[(10, 'a'), (5, 'b'), (2, 'c'), (8, 'd')]
>>>
>>> heapq.heapify(hq) # file de priorité
>>> hq
[(2, 'c'), (5, 'b'), (10, 'a'), (8, 'd')]
>>>
>>> heapq.heappush(hq, (1, 'e')) # Ajout du couple \
↪ {(1,'e')} dans hq
>>> hq
[(1, 'e'), (2, 'c'), (10, 'a'), (8, 'd'), (5, 'b')]
```

Exemple avec poids

```
>>> hq
[(1, 'e'), (2, 'c'), (10, 'a'), (8, 'd'), (5, 'b')]
>>>
>>> while len(hq) > 0: # Extraction des couples de hq
...     x = heapq.heappop(hq)
...     print(x)
...
(1, 'e')
(2, 'c')
(5, 'b')
(8, 'd')
(10, 'a')
```

Les sommets sont ici défilés, en respectant la règle de priorité de poids; les petits poids en premier.

Mise en oeuvre Python

- Le code de l'algorithme de Dijkstra dans le langage Python est similaire à celui de parcours en largeur.

Mise en oeuvre Python

- Le code de l'algorithme de Dijkstra dans le langage Python est similaire à celui de parcours en largeur.
- La file est remplacée par une file de priorité qui contient des couples (*distance au sommet de depart*, *sommet*). C'est le premier élément du couple qui définit la priorité.

Mise en oeuvre Python

- Le code de l'algorithme de Dijkstra dans le langage Python est similaire à celui de parcours en largeur.
- La file est remplacée par une file de priorité qui contient des couples (*distance au sommet de depart*, *sommet*). C'est le premier élément du couple qui définit la priorité.
- L'étape d'enfilement comporte un calcul lié au relâchement des arêtes.

Mise en oeuvre Python

- Le code de l'algorithme de Dijkstra dans le langage Python est similaire à celui de parcours en largeur.
- La file est remplacée par une file de priorité qui contient des couples (*distance au sommet de depart*, *sommet*). C'est le premier élément du couple qui définit la priorité.
- L'étape d'enfilement comporte un calcul lié au relâchement des arêtes.
- La fonction `dijkstra` ci-dessous reçoit un graphe `g` défini sous la forme d'un dictionnaire dont les clés sont les sommets et dont les valeurs sont les listes des arcs d'origine la clef, un arc étant un couple (*sommet de destination*, *poids*) et un sommet v_{init} à partir duquel sont recherchés les plus courts chemins.

Mise en oeuvre Python

- Le code de l'algorithme de Dijkstra dans le langage Python est similaire à celui de parcours en largeur.
- La file est remplacée par une file de priorité qui contient des couples (*distance au sommet de depart*, *sommet*). C'est le premier élément du couple qui définit la priorité.
- L'étape d'enfilement comporte un calcul lié au relâchement des arêtes.
- La fonction `dijkstra` ci-dessous reçoit un graphe `g` défini sous la forme d'un dictionnaire dont les clés sont les sommets et dont les valeurs sont les listes des arcs d'origine la clef, un arc étant un couple (*sommet de destination*, *poids*) et un sommet v_{init} à partir duquel sont recherchés les plus courts chemins.
- La fonction renvoie un couple formé du dictionnaire des distances minimales du sommet v_{init} aux sommets du graphe et du dictionnaire du prédécesseur de sommet u dans un chemin de longueur minimal de v_{init} à u.

CODE DE LA FONCTION dijkstra

```
def dijkstra(g, v_init):
    visited = {x : False for x in g}
    pred = {x : None for x in g}
    dist = {x : float('inf') for x in g}
    dist[v_init] = 0
    hq = [(0, v_init)]
    heapq.heapify(hq)
    while len(hq) > 0:
        dv, v = heapq.heappop(hq)
        if not visited[v]:
            visited[v] = True
            for w, dvw in g[v]:
                if not visited[w]:
                    dw = dv + dvw
                    if dw < dist[w]:
                        dist[w] = dw
                        pred[w] = v
                        heapq.heappush(hq, \
                            (dw, w))
    return dist,pred
```

dico des sommets visités
dico des predecesseurs
dico des distances
vinit est à distance 0 de lui-même

création de la FP
visite des sommets
extraction du sommet de prio min

parcours des voisins non visités de v

relâchement de l'arête (v,w)

maj de la distance min
maj du prédécesseur

ajout dans la FP

Application au graphe G_2

```
>>> g2 = {
...   'a' : [('b', 3), ('d', 9)],
...   'b' : [('a', 1), ('c', 4), ('e', 2)],
...   'c' : [('b', 1), ('f', 6)],
...   'd' : [('a', 7), ('e', 5), ('g', 9)],
...   'e' : [('b', 3), ('d', 9), ('f', 2), ('h', 8)],
...   'f' : [('c', 2), ('e', 6), ('i', 6)],
...   'g' : [('d', 3), ('h', 5)],
...   'h' : [('e', 4), ('g', 3), ('i', 5)],
...   'i' : [('f', 4), ('h', 8)]}
>>>
>>> dist, pred = dijkstra(g2, 'a')
>>> dist
{'a': 0, 'b': 3, 'c': 7, 'd': 9, 'e': 5, 'f': 7, 'g': 16, 'h': 13, 'i': 13}
>>> pred
{'a': None, 'b': 'a', 'c': 'b', 'd': 'a', 'e': 'b', 'f': 'e', 'g': 'h', 'h': \
↪ 'e', 'i': 'f'}
```

Complexité

- Pour un graphe $G = (S, A)$, notons $|S|$ et $|A|$ les nombres de sommets et d'arêtes.

Complexité

- Pour un graphe $G = (S, A)$, notons $|S|$ et $|A|$ les nombres de sommets et d'arêtes.
- **Coût de l'initialisation.** La construction des trois premiers dictionnaires est de coût $O(|S|)$, à chaque fois en raison de la boucle sur les sommets du graphe. La création de la file de priorité est ici en $O(1)$.

Complexité

- Pour un graphe $G = (S, A)$, notons $|S|$ et $|A|$ les nombres de sommets et d'arêtes.
- **Coût de l'initialisation.** La construction des trois premiers dictionnaires est de coût $O(|S|)$, à chaque fois en raison de la boucle sur les sommets du graphe. La création de la file de priorité est ici en $O(1)$.
- **Coût du parcours.** L'algorithme visite chaque arc au plus une fois. Chaque considération d'un arc peut conduire à l'ajout d'un élément dans la file. La file de priorité peut donc contenir jusqu'à $|A|$ éléments. Comme écrit plus haut, les opérations d'ajout et d'extraction ont un coût logarithmique. Chaque opération sur la file a donc un coût $O(\log|A|)$. Or $|A| \leqslant |S|^2$ de sorte que $\log|A| = O(\log|S|)$.

- Pour un graphe $G = (S, A)$, notons $|S|$ et $|A|$ les nombres de sommets et d'arêtes.
- **Coût de l'initialisation.** La construction des trois premiers dictionnaires est de coût $O(|S|)$, à chaque fois en raison de la boucle sur les sommets du graphe. La création de la file de priorité est ici en $O(1)$.
- **Coût du parcours.** L'algorithme visite chaque arc au plus une fois. Chaque considération d'un arc peut conduire à l'ajout d'un élément dans la file. La file de priorité peut donc contenir jusqu'à $|A|$ éléments. Comme écrit plus haut, les opérations d'ajout et d'extraction ont un coût logarithmique. Chaque opération sur la file a donc un coût $O(\log |A|)$. Or $|A| \leqslant |S|^2$ de sorte que $\log |A| = O(\log |S|)$.
- La complexité de `dijkstra` est donc en $O(|A| \log |S|)$ ou encore en $O(|S|^2 \log |S|)$.

Application à la recherche de plus court chemin

- La fonction `dijkstra(g,v_init)` peut être transformée (c'est l'objet d'un exercice du TP) en une fonction `dijkstra_path(g,v_init,v_fin)` prenant en entrée deux sommets `v_init` et `v_fin` d'un graphe G codé par le dictionnaire `g` et qui :

- La fonction `dijkstra(g,v_init)` peut être transformée (c'est l'objet d'un exercice du TP) en une fonction `dijkstra_path(g,v_init,v_fin)` prenant en entrée deux sommets `v_init` et `v_fin` d'un graphe G codé par le dictionnaire `g` et qui :
    - ◇ lorsque `v_fin` n'est pas accessible depuis `v_init`, affiche un message l'indiquant,

Application à la recherche de plus court chemin

- La fonction `dijkstra(g,v_init)` peut être transformée (c'est l'objet d'un exercice du TP) en une fonction `dijkstra_path(g,v_init,v_fin)` prenant en entrée deux sommets `v_init` et `v_fin` d'un graphe G codé par le dictionnaire `g` et qui :
  - lorsque `v_fin` n'est pas accessible depuis `v_init`, affiche un message l'indiquant,
  - lorsque `v_fin` est accessible depuis `v_init`, renvoie le triplet `(N,d,C)` où :

- La fonction `dijkstra(g,v_init)` peut être transformée (c'est l'objet d'un exercice du TP) en une fonction `dijkstra_path(g,v_init,v_fin)` prenant en entrée deux sommets `v_init` et `v_fin` d'un graphe G codé par le dictionnaire `g` et qui :
  - ◇ lorsque `v_fin` n'est pas accessible depuis `v_init`, affiche un message l'indiquant,
  - ◇ lorsque `v_fin` est accessible depuis `v_init`, renvoie le triplet `(N,d,C)` où :
    - — `N` est le nombre de sommets qui ont été visités pour détecter un plus court chemin,

Application à la recherche de plus court chemin

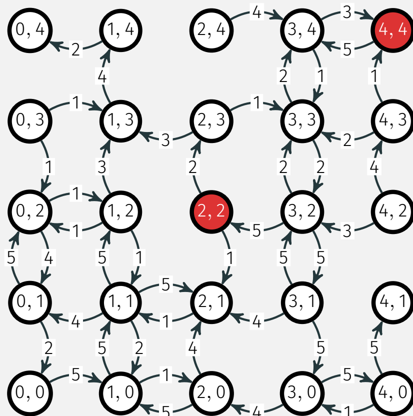
- La fonction `dijkstra(g,v_init)` peut être transformée (c'est l'objet d'un exercice du TP) en une fonction `dijkstra_path(g,v_init,v_fin)` prenant en entrée deux sommets `v_init` et `v_fin` d'un graphe G codé par le dictionnaire `g` et qui :
  - ◇ lorsque `v_fin` n'est pas accessible depuis `v_init`, affiche un message l'indiquant,
  - ◇ lorsque `v_fin` est accessible depuis `v_init`, renvoie le triplet `(N,d,C)` où :
    - — `N` est le nombre de sommets qui ont été visités pour détecter un plus court chemin,
    - — `d` est la distance de `v_init` à `v_fin`,

- La fonction `dijkstra(g,v_init)` peut être transformée (c'est l'objet d'un exercice du TP) en une fonction `dijkstra_path(g,v_init,v_fin)` prenant en entrée deux sommets `v_init` et `v_fin` d'un graphe G codé par le dictionnaire `g` et qui :
  - ◇ lorsque `v_fin` n'est pas accessible depuis `v_init`, affiche un message l'indiquant,
  - ◇ lorsque `v_fin` est accessible depuis `v_init`, renvoie le triplet `(N,d,C)` où :
    - — `N` est le nombre de sommets qui ont été visités pour détecter un plus court chemin,
    - — `d` est la distance de `v_init` à `v_fin`,
    - — `C` est un meilleur chemin, i.e. une liste de sommets, menant de `v_init` à `v_fin`.

Application à la recherche de plus court chemin

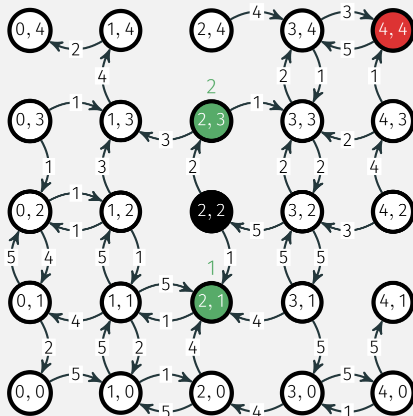
- La fonction `dijkstra(g,v_init)` peut être transformée (c'est l'objet d'un exercice du TP) en une fonction `dijkstra_path(g,v_init,v_fin)` prenant en entrée deux sommets `v_init` et `v_fin` d'un graphe G codé par le dictionnaire `g` et qui :
  - ◇ lorsque `v_fin` n'est pas accessible depuis `v_init`, affiche un message l'indiquant,
  - ◇ lorsque `v_fin` est accessible depuis `v_init`, renvoie le triplet `(N,d,C)` où :
    - — `N` est le nombre de sommets qui ont été visités pour détecter un plus court chemin,
    - — `d` est la distance de `v_init` à `v_fin`,
    - — `C` est un meilleur chemin, i.e. une liste de sommets, menant de `v_init` à `v_fin`.
- Les diapos suivantes illustrent cet algorithme de recherche (blanc : sommet ni visité ni découvert, noir : sommet visité, bleu : sommet découvert non visité avec leur distance au sommet initial (au-dessus du sommet), rouge : sommet du meilleur chemin détecté).

Exemple de recherche de plus court chemin



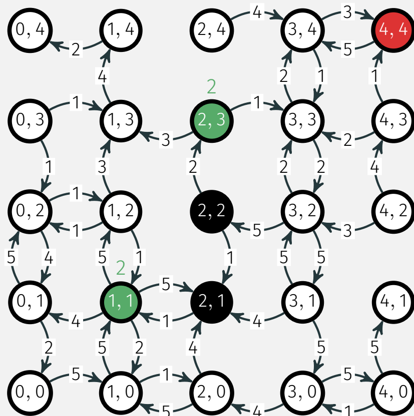
Initialisation

Exemple de recherche de plus court chemin



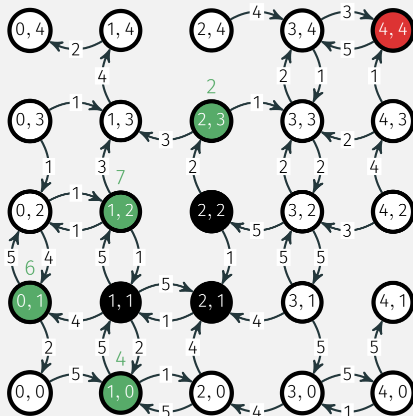
Etape 1

Exemple de recherche de plus court chemin



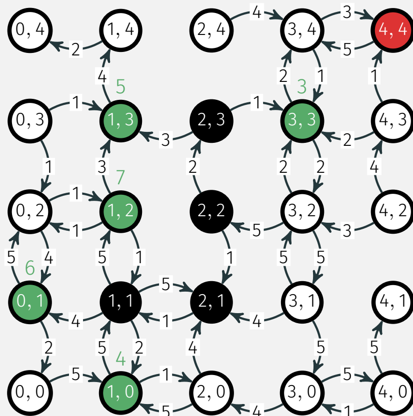
Etape 2

Exemple de recherche de plus court chemin



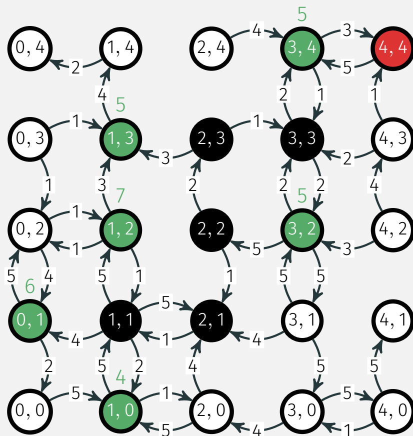
Etape 3

Exemple de recherche de plus court chemin



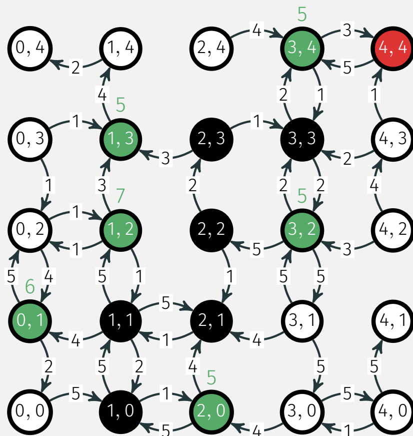
Etape 4

Exemple de recherche de plus court chemin

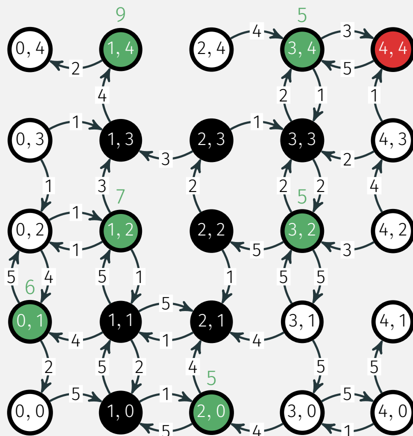


Etape 5

Exemple de recherche de plus court chemin



Etape 6

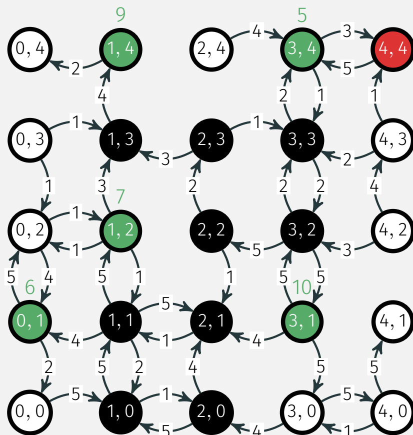


Etape 7

Exemple de recherche de plus court chemin

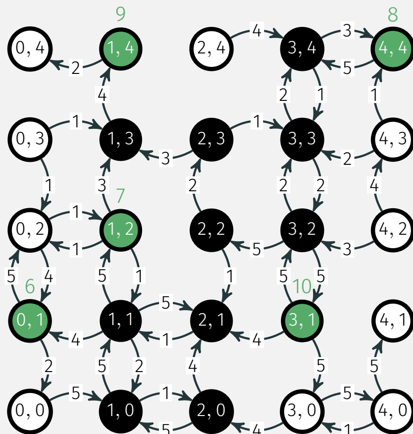
Etape 8

Exemple de recherche de plus court chemin



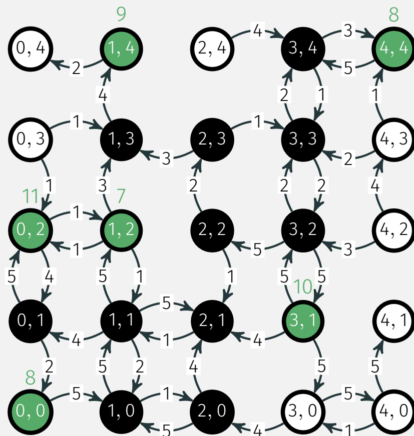
Etape 9

Exemple de recherche de plus court chemin

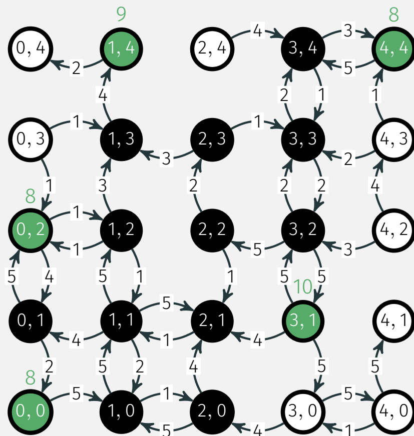


Etape 10

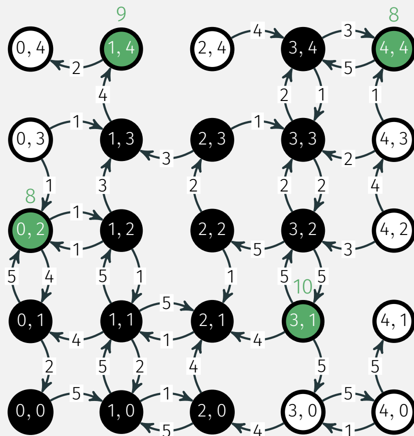
Exemple de recherche de plus court chemin



Etape 11



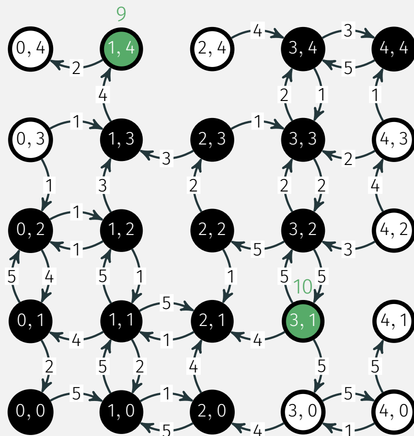
Etape 12



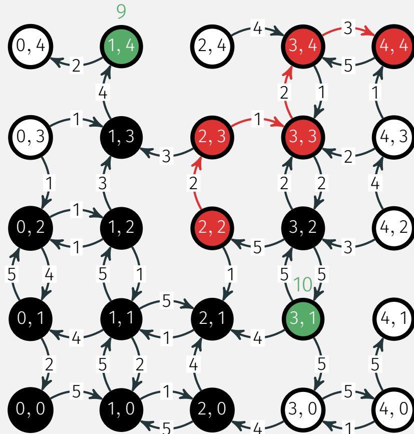
Etape 13

Exemple de recherche de plus court chemin

Etape 14



Etape 15



Fin du parcours :

- 15 sommets visités,
- distance obtenue : 8,
- 10 sommets visités inutilement.

Inconvénients de `dijkstra_path(g,v_init,v_fin)`

- L'algorithme visite tous les sommets par ordre de distance croissant depuis `v_init` jusqu'à rencontrer `v_fin`, cela conduit donc à visiter inutilement beaucoup de sommets.

INCONVÉNIENTS DE `dijkstra_path(g,v_init,v_fin)`

- L'algorithme visite tous les sommets par ordre de distance croissant depuis `v_init` jusqu'à rencontrer `v_fin`, cela conduit donc à visiter inutilement beaucoup de sommets.
- Idée d'amélioration : forcer l'algorithme à prioriser l'étude des sommets qui “sont dans la bonne direction”. Pour cela, et puisque c'est les sommets de priorité minimale qui sont traités en premier, il faudrait dans la file de priorité :

INCONVÉNIENTS DE `dijkstra_path(g,v_init,v_fin)`

- L'algorithme visite tous les sommets par ordre de distance croissant depuis `v_init` jusqu'à rencontrer `v_fin`, cela conduit donc à visiter inutilement beaucoup de sommets.
- Idée d'amélioration : forcer l'algorithme à prioriser l'étude des sommets qui "sont dans la bonne direction". Pour cela, et puisque c'est les sommets de priorité minimale qui sont traités en premier, il faudrait dans la file de priorité :
  - ◇ diminuer la priorité des sommets qui semblent se rapprocher `v_fin`,

INCONVÉNIENTS DE `dijkstra_path(g,v_init,v_fin)`

- L'algorithme visite tous les sommets par ordre de distance croissant depuis `v_init` jusqu'à rencontrer `v_fin`, cela conduit donc à visiter inutilement beaucoup de sommets.
- Idée d'amélioration : forcer l'algorithme à prioriser l'étude des sommets qui "sont dans la bonne direction". Pour cela, et puisque c'est les sommets de priorité minimale qui sont traités en premier, il faudrait dans la file de priorité :
  - ◇ diminuer la priorité des sommets qui semblent se rapprocher `v_fin`,
  - ◇ augmenter la priorité des sommets qui semblent s'en éloigner.

INCONVÉNIENTS DE `dijkstra_path(g,v_init,v_fin)`

- L'algorithme visite tous les sommets par ordre de distance croissant depuis `v_init` jusqu'à rencontrer `v_fin`, cela conduit donc à visiter inutilement beaucoup de sommets.
- Idée d'amélioration : forcer l'algorithme à prioriser l'étude des sommets qui "sont dans la bonne direction". Pour cela, et puisque c'est les sommets de priorité minimale qui sont traités en premier, il faudrait dans la file de priorité :
  - diminuer la priorité des sommets qui semblent se rapprocher `v_fin`,
  - augmenter la priorité des sommets qui semblent s'en éloigner.
- Il faut donc pouvoir quantifier la *proximité* d'un sommet à `v_fin`, c'est-à-dire donner une estimation de ce qu'il reste à parcourir. Nous allons le mettre en place avec la notion d'*heuristique* et l'algorithme A*.

Algorithme A*

Principe de A*

- Une *heuristique* est une fonction h qui permet d'estimer la proximité de deux sommets : $h(v, w)$ est le *coût estimé* du chemin le moins coûteux de v à w.

Principe de A*

- Une *heuristique* est une fonction h qui permet d'estimer la proximité de deux sommets : $h(v, w)$ est le *coût estimé* du chemin le moins coûteux de v à w.
- L'algorithme A* reprend l'algorithme de Dijkstra en modifiant le calcul de la priorité p_w d'un sommet w : elle n'est plus simplement sa distance depuis `v_init` mais sa distance depuis `v_init` à laquelle on ajoute la valeur de son heuristique vers `v_fin`, i.e une estimation du coût de ce qu'il reste à parcourir :

$$p_w = d_{v_{init}}[w] + \mathrm{h}(\mathrm{w}, \mathrm{v}_{\mathrm{fin}}).$$

Principe de A*

- Une *heuristique* est une fonction h qui permet d'estimer la proximité de deux sommets : $h(v, w)$ est le *coût estimé* du chemin le moins coûteux de v à w.
- L'algorithme A* reprend l'algorithme de Dijkstra en modifiant le calcul de la priorité p_w d'un sommet w : elle n'est plus simplement sa distance depuis `v_init` mais sa distance depuis `v_init` à laquelle on ajoute la valeur de son heuristique vers `v_fin`, i.e une estimation du coût de ce qu'il reste à parcourir :

$$p_w = d_{v_{init}}[w] + \mathrm{h}(\mathrm{w}, \mathrm{v_{fin}}).$$

- Plusieurs heuristique sont possibles, l'efficacité de l'algorithme A* étant conditionnée au choix d'une heuristique adaptée à la situation.

Principe de A*

- Une *heuristique* est une fonction h qui permet d'estimer la proximité de deux sommets : $h(v, w)$ est le *coût estimé* du chemin le moins coûteux de v à w.
- L'algorithme A* reprend l'algorithme de Dijkstra en modifiant le calcul de la priorité p_w d'un sommet w : elle n'est plus simplement sa distance depuis `v_init` mais sa distance depuis `v_init` à laquelle on ajoute la valeur de son heuristique vers `v_fin`, i.e une estimation du coût de ce qu'il reste à parcourir :

$$p_w = d_{v_{init}}[w] + \mathrm{h}(\mathrm{w}, \mathrm{v}_{\mathrm{fin}}).$$

- Plusieurs heuristique sont possibles, l'efficacité de l'algorithme A* étant conditionnée au choix d'une heuristique adaptée à la situation.
- La prochaine diapo contient le code de la fonction `a_star_path(g, v_init,v_fin, h)` (`v_fin` supposé accessible) où l'heuristique est codée sous la forme d'un dictionnaire (clé : sommets, valeur : heuristique du sommet à `v_fin`).

```
def a_star_path(g, v_init,v_fin, h):
    visited = {x : False for x in g}
    pred = {x : None for x in g}
    dist = {x : float('inf') for x in g}
    dist[v_init] = 0
    hq,N,C = [(h[v_init], v_init)],0,[v_fin]
    heapq.heapify(hq)
    while len(hq) > 0 and not visited[v_fin]:
        pv,v = heapq.heappop(hq)
        if not visited[v]:
            visited[v], N = True, N+1
            for w, dvw in g[v]:
                if not visited[w]:
                    dw = dist[v]+dvw
                    pw = dist[v]+dvw+h[w]
                    if dw < dist[w]:
                        dist[w], pred[w] = dw,v
                        heapq.heappush(hq, (pw, w))
    if not visited[v_fin]:
        print("Pas de chemin de "\
            +str(v_init)+" à "+str(v_fin))
      else:
        while C[0] != v_init:
        w = pred[C[0]]
        C.append(w)
      return N,dist[v_fin], C
```

dico des sommets visités

dico des prédécesseurs

dico des dist

FP, compteur des sommets vis.,chemin

extraction du sommet de prio min

maj du compteur

parcours des vois. non visités de v

maj de la dist et du pred

stockage dans la FP

construction du chemin

- Nous allons illustrer le déroulement de Dijkstra et de A* sur différents graphes et pour différentes heuristiques.

Illustrations

- Nous allons illustrer le déroulement de Dijkstra et de A* sur différents graphes et pour différentes heuristiques.
- L’étiquette d’un sommet sera ses coordonnées dans le plan.

- Nous allons illustrer le déroulement de Dijkstra et de A* sur différents graphes et pour différentes heuristiques.
- L'étiquette d'un sommet sera ses coordonnées dans le plan.
- Nous utiliserons comme heuristiques les distances suivantes du plan réel : soit $u(x_1, y_1)$ et $v(x_2, y_2)$,

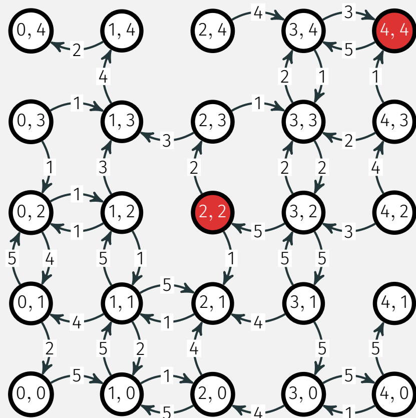
ILLUSTRATIONS

- Nous allons illustrer le déroulement de DIJKSTRA et de A* sur différents graphes et pour différentes heuristiques.
- L'étiquette d'un sommet sera ses coordonnées dans le plan.
- Nous utiliserons comme heuristiques les distances suivantes du plan réel : soit $u(x_1, y_1)$ et $v(x_2, y_2)$,
  - ◇ $d_p(u, v) = (|x_2 - x_1|^p + |y_2 - y_1|^p)^{\frac{1}{p}}, \quad \forall p \in [1, +\infty[$

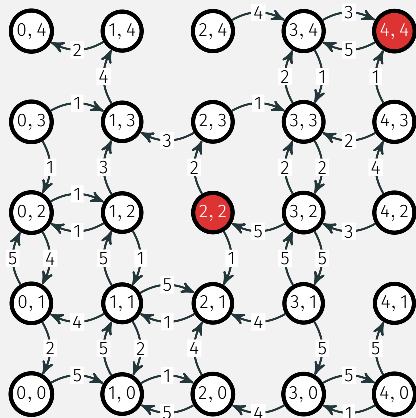
- Nous allons illustrer le déroulement de Dijkstra et de A* sur différents graphes et pour différentes heuristiques.
- L'étiquette d'un sommet sera ses coordonnées dans le plan.
- Nous utiliserons comme heuristiques les distances suivantes du plan réel : soit $u(x_1, y_1)$ et $v(x_2, y_2)$,
  - $d_p(u, v) = \left(|x_2 - x_1|^p + |y_2 - y_1|^p\right)^{\frac{1}{p}}, \quad \forall p \in [1, +\infty[$
  - $d_\infty(u, v) = \max\left(|x_2 - x_1|, |y_2 - y_1|\right)$

- Nous allons illustrer le déroulement de DIJKSTRA et de A* sur différents graphes et pour différentes heuristiques.
- L'étiquette d'un sommet sera ses coordonnées dans le plan.
- Nous utiliserons comme heuristiques les distances suivantes du plan réel : soit $u(x_1, y_1)$ et $v(x_2, y_2)$,
  - $d_p(u, v) = (|x_2 - x_1|^p + |y_2 - y_1|^p)^{\frac{1}{p}}, \quad \forall p \in [1, +\infty[$
  - $d_\infty(u, v) = \max(|x_2 - x_1|, |y_2 - y_1|)$
- Au-dessus des sommets découverts et non-visités (en bleu) est indiqué soit la distance au sommet initial (DIJKSTRA), soit la distance au sommet initial + l'arrondi à l'entier le plus proche de l'heuristique (A*).

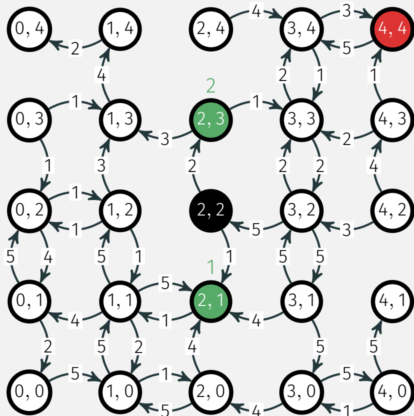
Ex1 : A* converge plus vite vers un aussi bon chemin



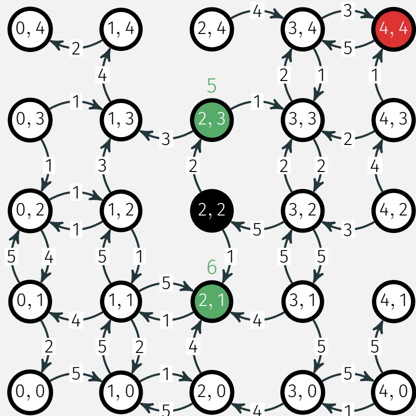
Dijkstra, initialisation



A* heuristique d_1, initialisation

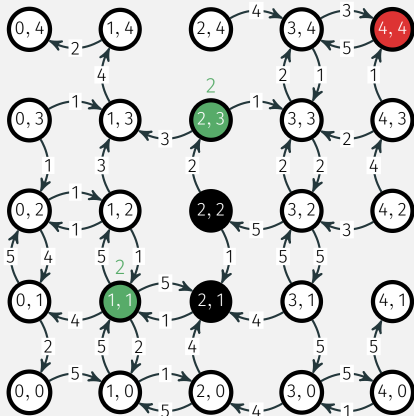


Dijkstra, étape 1

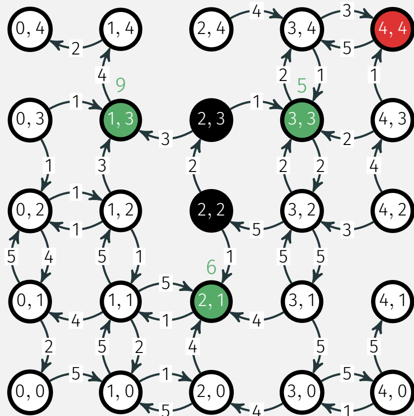


A* heuristique d_1, étape 1

Ex1 : A* converge plus vite vers un aussi bon chemin

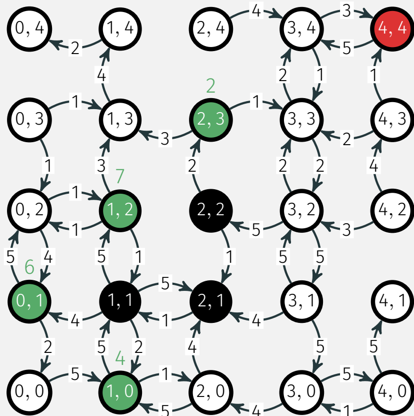


Dijkstra, étape 2

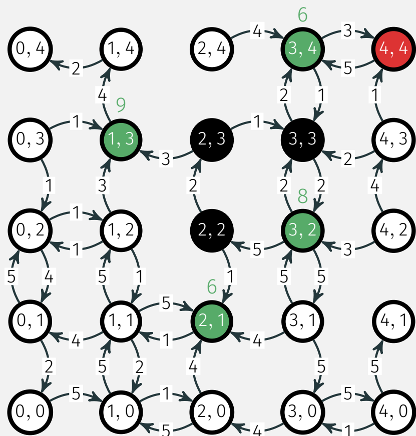


A* heuristique d_1, étape 2

Ex1 : A* converge plus vite vers un aussi bon chemin

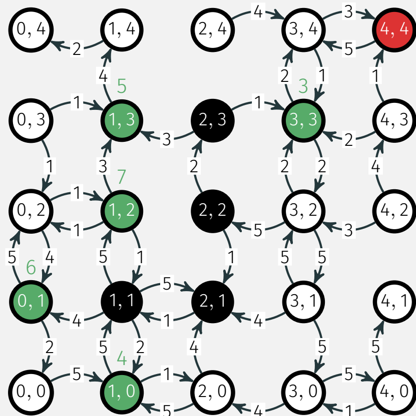


Dijkstra, étape 3

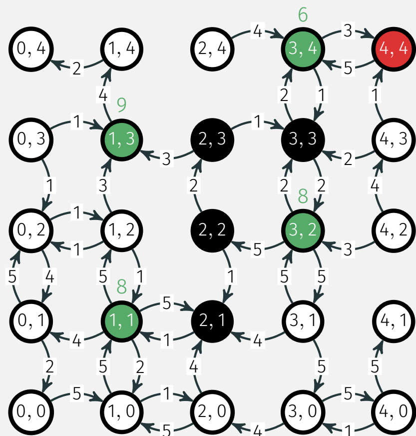


A* heuristique d_1, étape 3

Ex1 : A* converge plus vite vers un aussi bon chemin

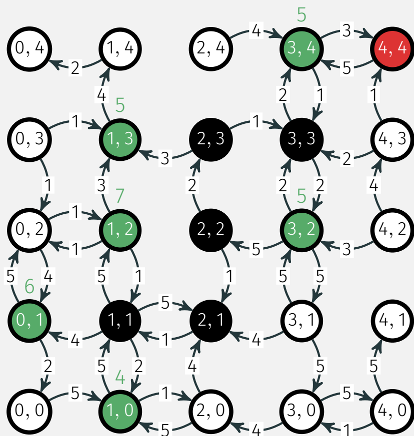


Dijkstra, étape 4

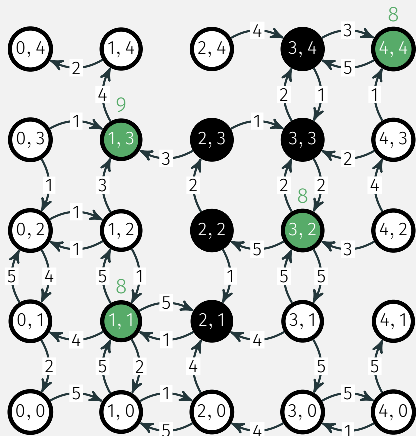


A* heuristique d_1, étape 4

Ex1 : A* converge plus vite vers un aussi bon chemin

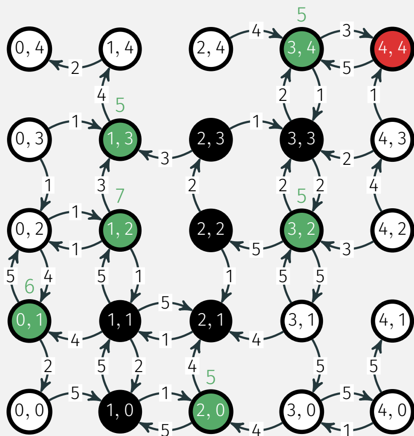


Dijkstra, étape 5

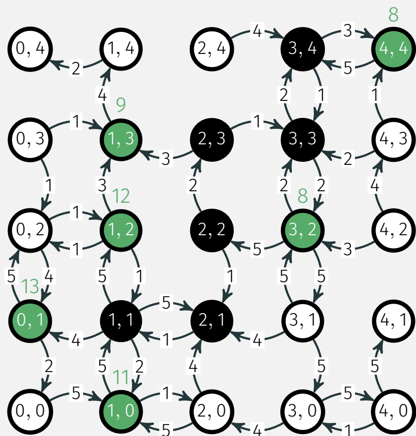


A* heuristique d_1, étape 5

Ex1 : A* converge plus vite vers un aussi bon chemin

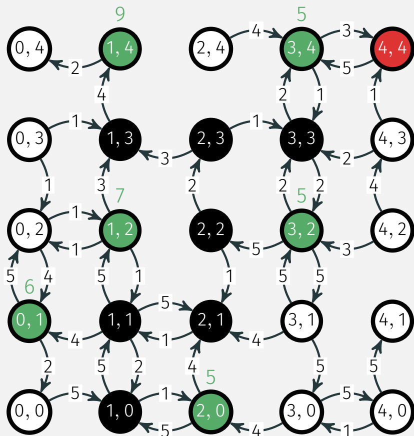


Dijkstra, étape 6

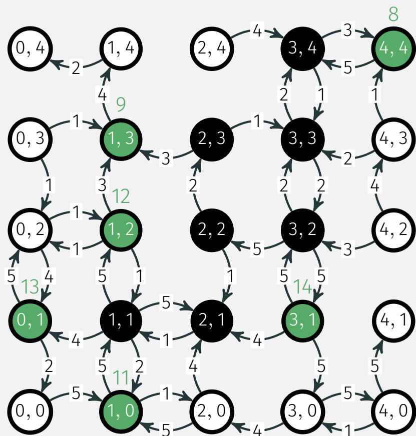


A* heuristique d_1, étape 6

Ex1 : A* converge plus vite vers un aussi bon chemin

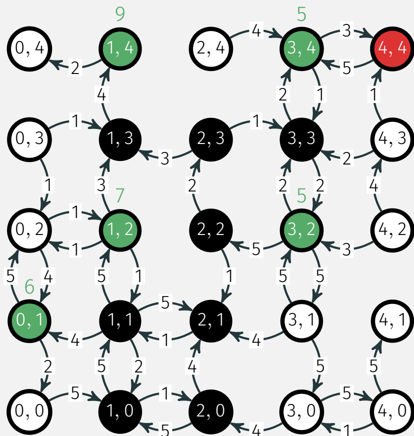


Dijkstra, étape 7

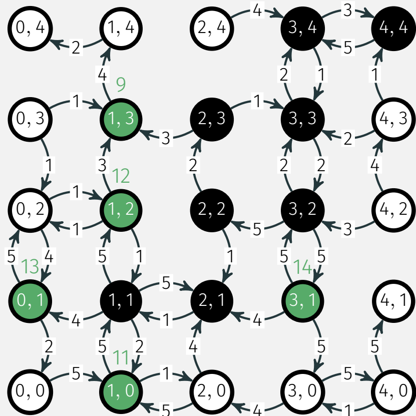


A* heuristique d_1, étape 7

Ex1 : A* converge plus vite vers un aussi bon chemin

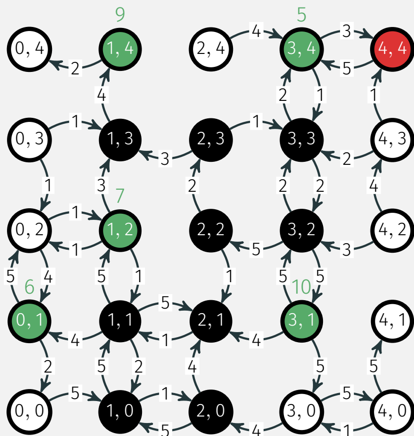


Dijkstra, étape 8

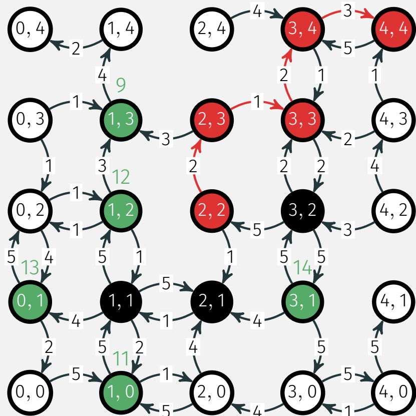


A* heuristique d_1, étape 8

EX1 : A* CONVERGE PLUS VITE VERS UN AUSSI BON CHEMIN



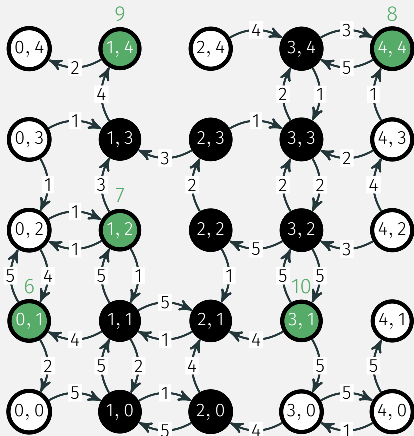
DIJKSTRA, étape 9



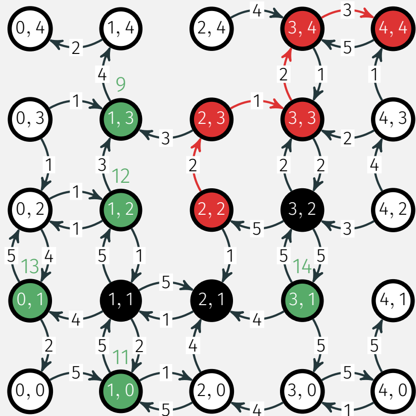
A* heuristique d_1, fin du parcours :

- 8 sommets visités,
- distance obtenue : 8,
- 3 sommets visités inutilement.

Ex1 : A* converge plus vite vers un aussi bon chemin



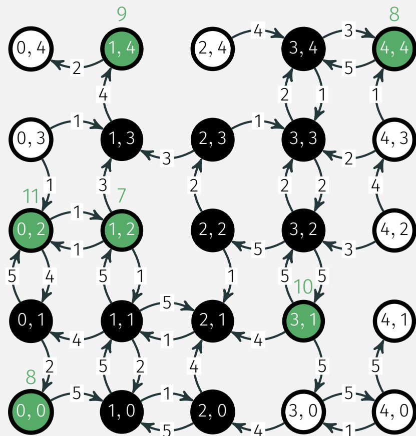
Dijkstra, étape 10



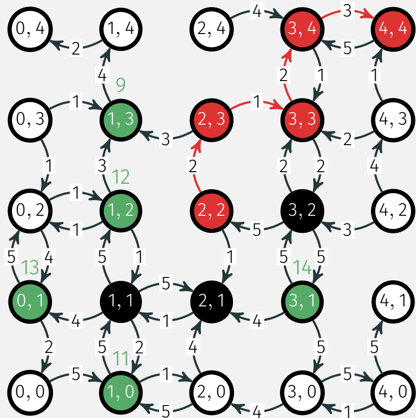
A* heuristique d_1, fin du parcours :

- 8 sommets visités,
- distance obtenue : 8,
- 3 sommets visités inutilement.

Ex1 : A* converge plus vite vers un aussi bon chemin



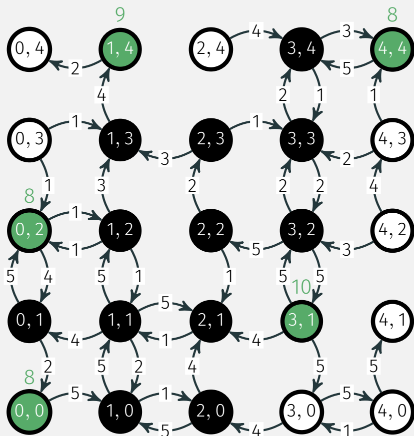
Dijkstra, étape 11



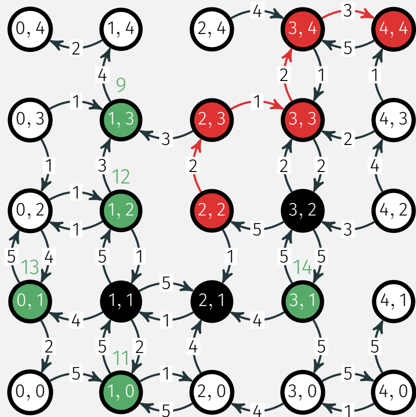
A* heuristique d_1, fin du parcours :

- 8 sommets visités,
- distance obtenue : 8,
- 3 sommets visités inutilement.

Ex1 : A* converge plus vite vers un aussi bon chemin



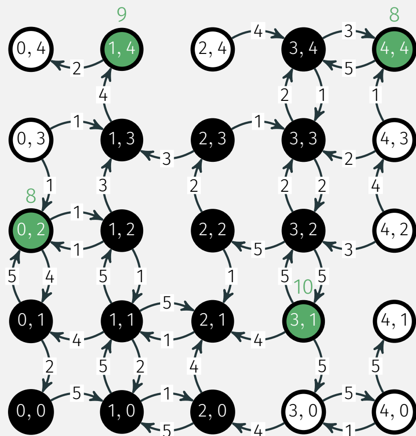
Dijkstra, étape 12



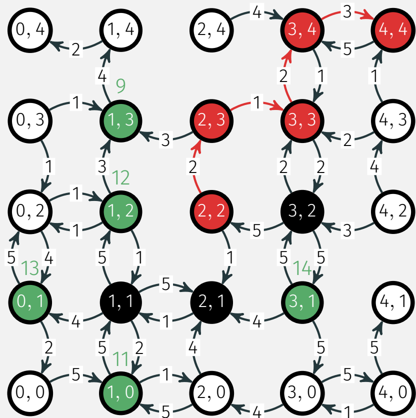
A* heuristique d_1, fin du parcours :

- 8 sommets visités,
- distance obtenue : 8,
- 3 sommets visités inutilement.

Ex1 : A* converge plus vite vers un aussi bon chemin



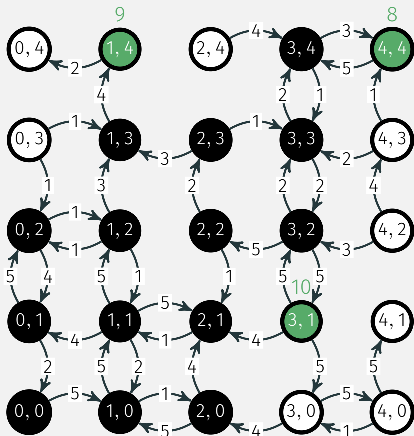
Dijkstra, étape 13



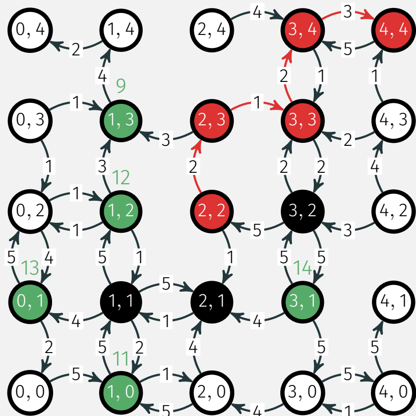
A* heuristique d_1, fin du parcours :

- 8 sommets visités,
- distance obtenue : 8,
- 3 sommets visités inutilement.

Ex1 : A* converge plus vite vers un aussi bon chemin



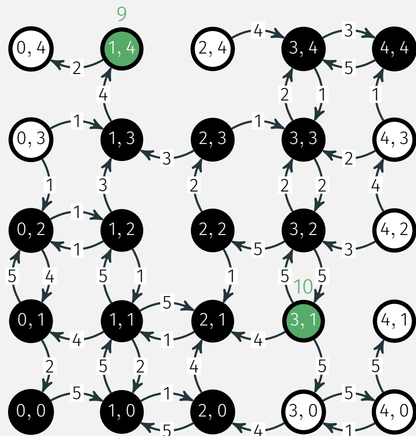
Dijkstra, étape 14



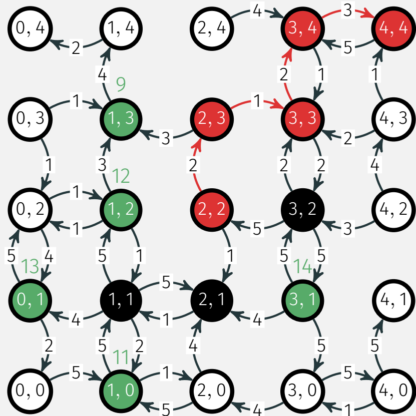
A* heuristique d_1, fin du parcours :

- 8 sommets visités,
- distance obtenue : 8,
- 3 sommets visités inutilement.

Ex1 : A* converge plus vite vers un aussi bon chemin



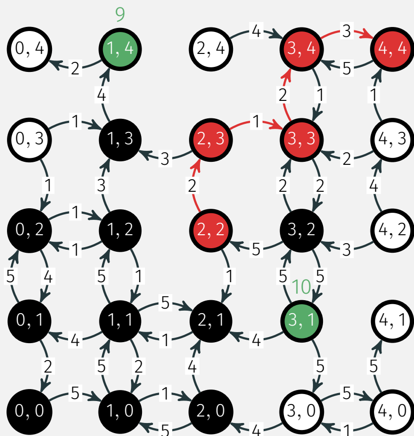
Dijkstra, étape 15



A* heuristique d_1, fin du parcours :

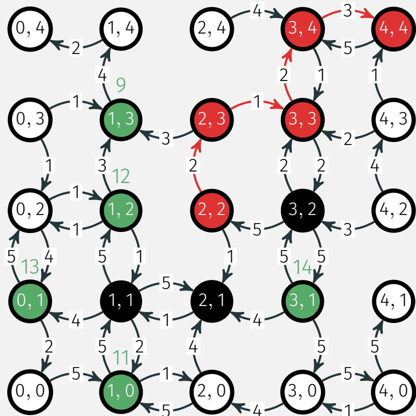
- 8 sommets visités,
- distance obtenue : 8,
- 3 sommets visités inutilement.

Ex1 : A* converge plus vite vers un aussi bon chemin



Dijkstra, fin du parcours :

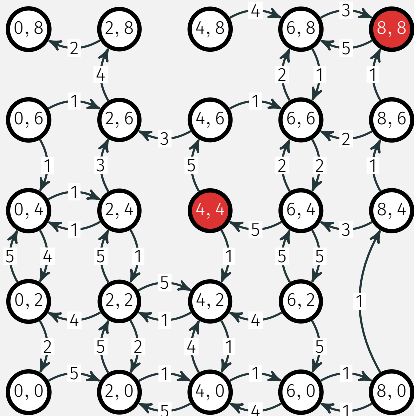
- 15 sommets visités,
- distance obtenue : 8,
- 10 sommets visités inutilement.



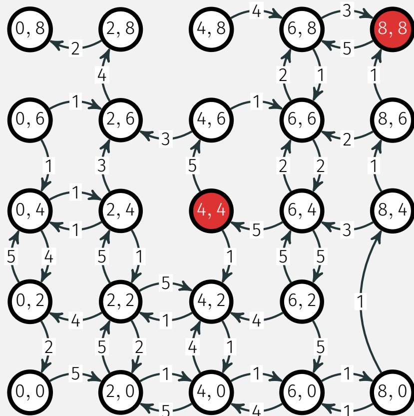
A* heuristique d_1, fin du parcours :

- 8 sommets visités,
- distance obtenue : 8,
- 3 sommets visités inutilement.

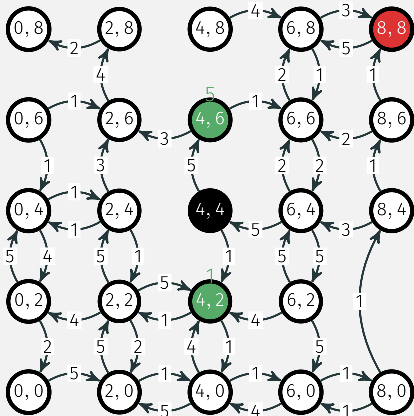
Ex2 : A* converge plus vite vers un moins bon chemin



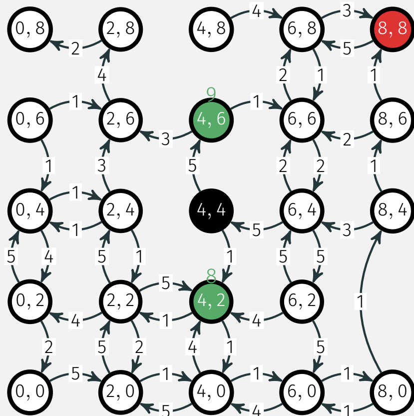
Dijkstra, initialisation



A* heuristique d_2, initialisation

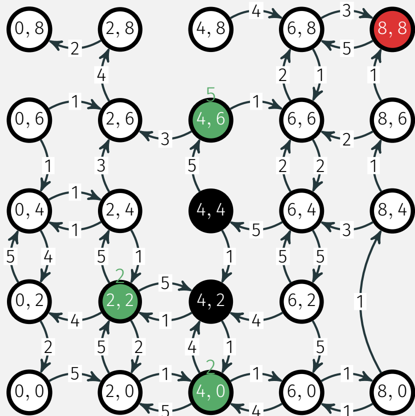


Dijkstra, étape 1

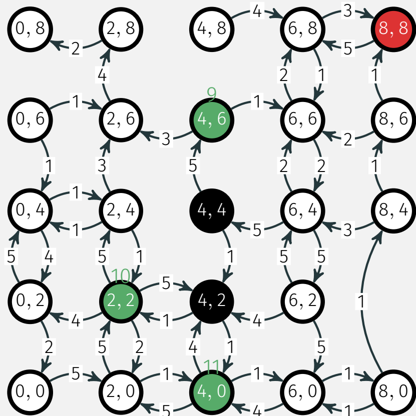


A* heuristique d_2, étape 1

Ex2 : A* converge plus vite vers un moins bon chemin

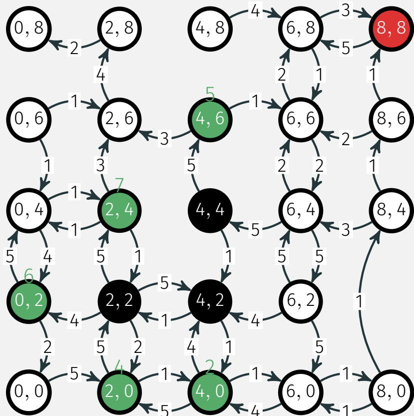


Dijkstra, étape 2

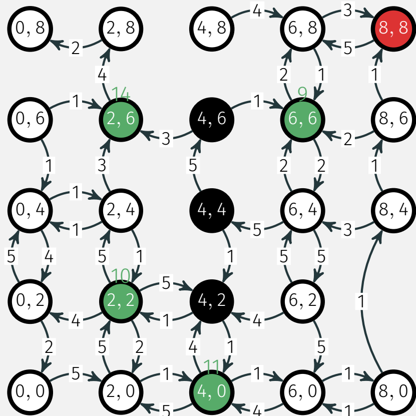


A* heuristique d_2, étape 2

Ex2 : A* converge plus vite vers un moins bon chemin

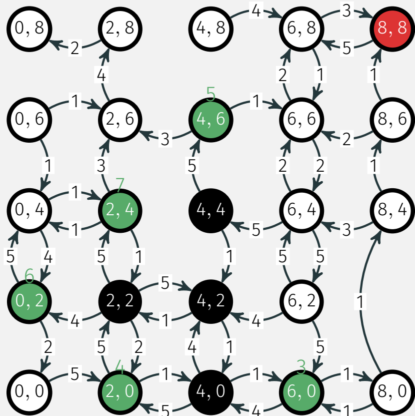


Dijkstra, étape 3

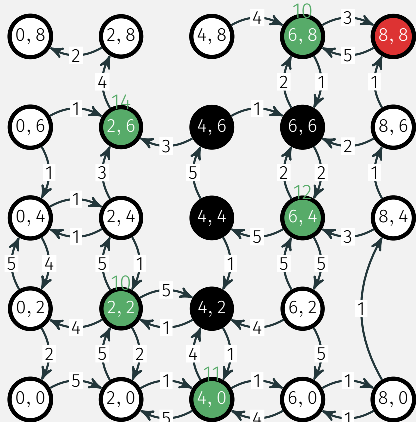


A* heuristique d_2, étape 3

EX2 : A* CONVERGE PLUS VITE VERS UN MOINS BON CHEMIN

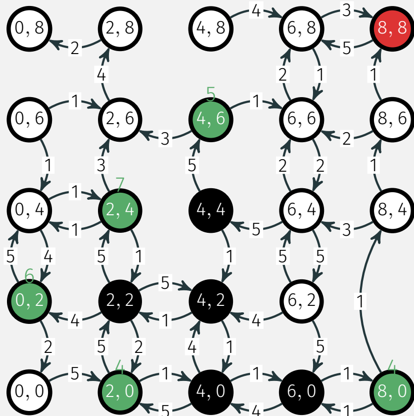


DIJKSTRA, étape 4

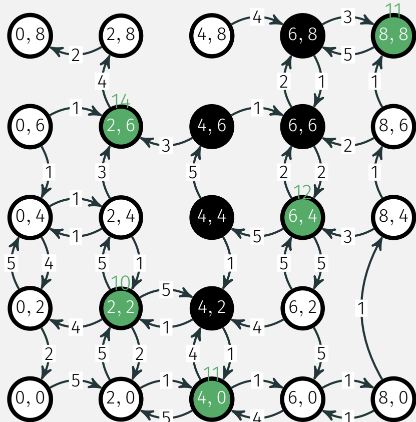


A* heuristique d_2, étape 4

Ex2 : A* converge plus vite vers un moins bon chemin

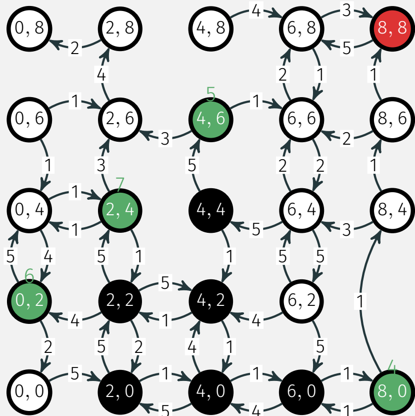


Dijkstra, étape 5

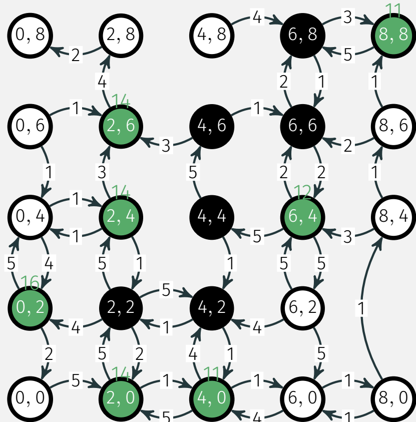


A* heuristique d_2, étape 5

Ex2 : A* converge plus vite vers un moins bon chemin

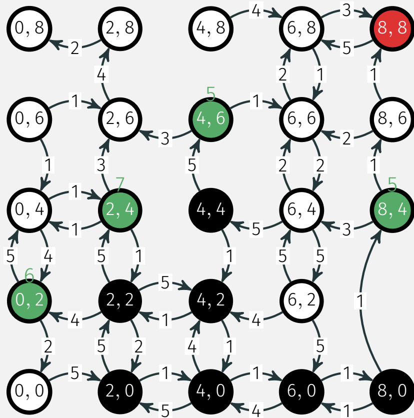


Dijkstra, étape 6

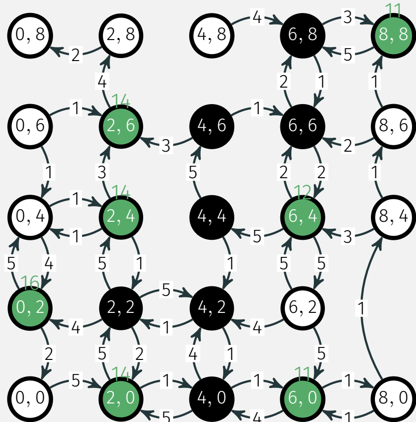


A* heuristique d_2, étape 6

Ex2 : A* converge plus vite vers un moins bon chemin

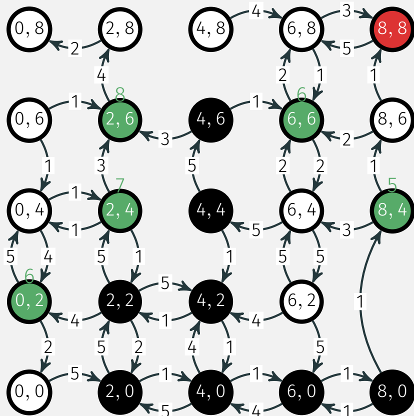


Dijkstra, étape 7

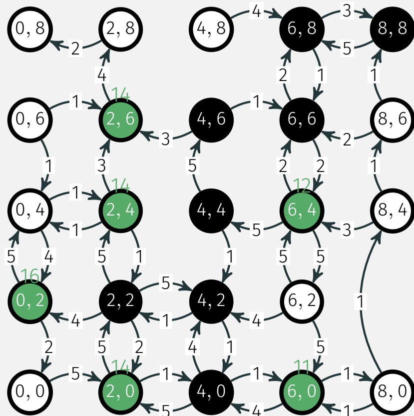


A* heuristique d_2, étape 7

Ex2 : A* converge plus vite vers un moins bon chemin

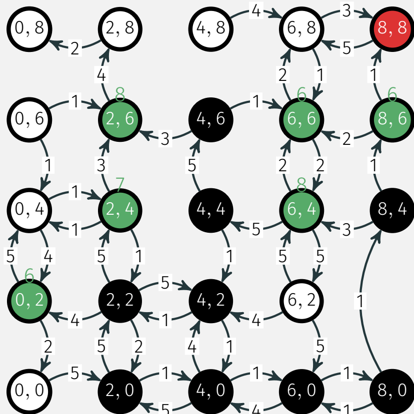


Dijkstra, étape 8

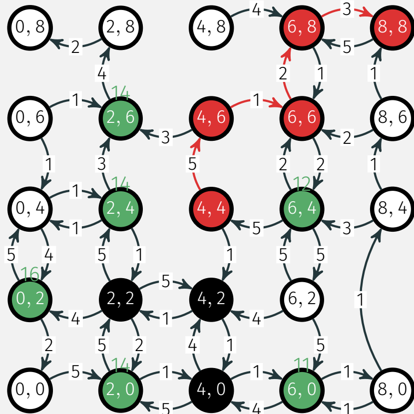


A* heuristique d_2, étape 8

Ex2 : A* converge plus vite vers un moins bon chemin



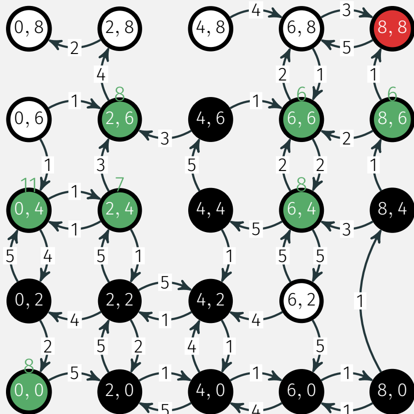
Dijkstra, étape 9



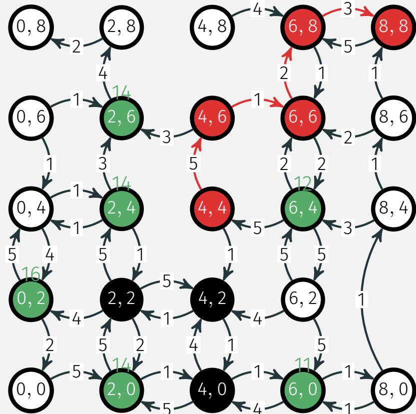
A* heuristique d_2, fin du parcours :

- 8 sommets visités,
- distance obtenue : 11,
- 3 sommets visités inutilement.

Ex2 : A* converge plus vite vers un moins bon chemin



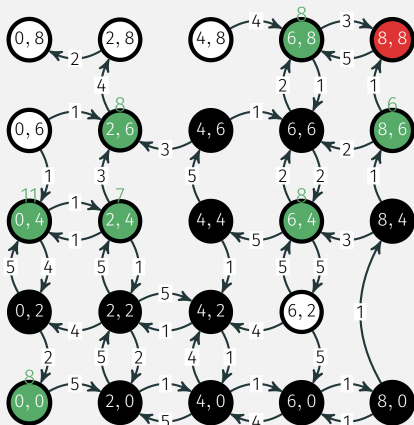
Dijkstra, étape 10



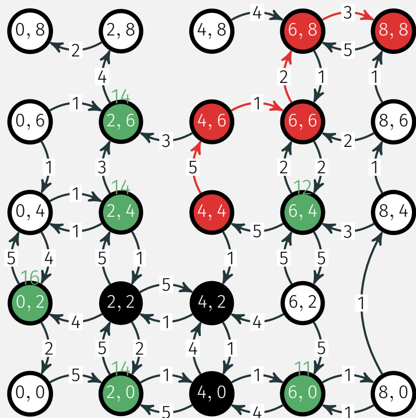
A* heuristique d_2, fin du parcours :

- 8 sommets visités,
- distance obtenue : 11,
- 3 sommets visités inutilement.

Ex2 : A* converge plus vite vers un moins bon chemin



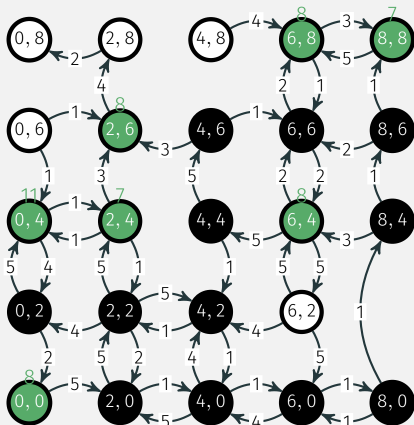
Dijkstra, étape 11



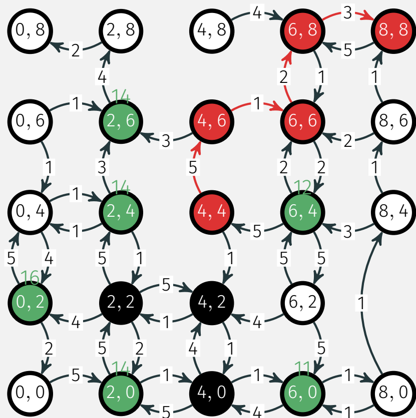
A* heuristique d_2, fin du parcours :

- 8 sommets visités,
- distance obtenue : 11,
- 3 sommets visités inutilement.

Ex2 : A* converge plus vite vers un moins bon chemin



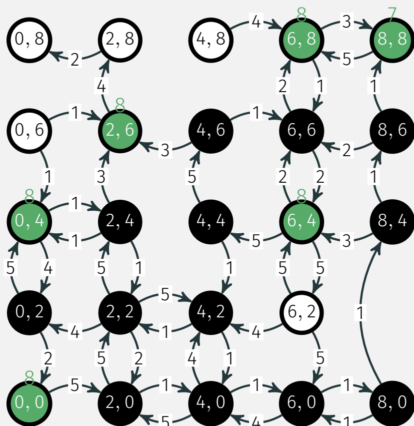
Dijkstra, étape 12



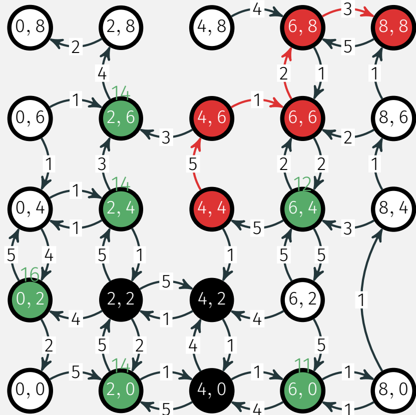
A* heuristique d_2, fin du parcours :

- 8 sommets visités,
- distance obtenue : 11,
- 3 sommets visités inutilement.

Ex2 : A* converge plus vite vers un moins bon chemin



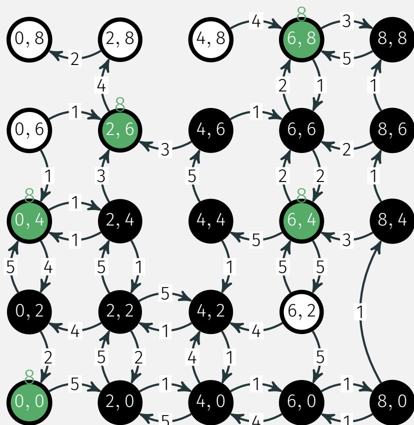
Dijkstra, étape 13



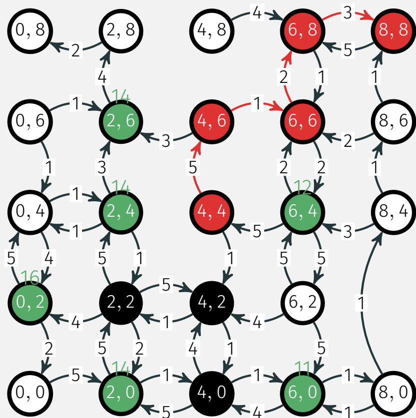
A* heuristique d_2, fin du parcours :

- 8 sommets visités,
- distance obtenue : 11,
- 3 sommets visités inutilement.

Ex2 : A* converge plus vite vers un moins bon chemin



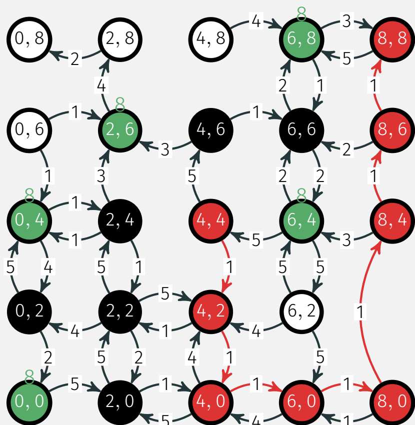
Dijkstra, étape 14

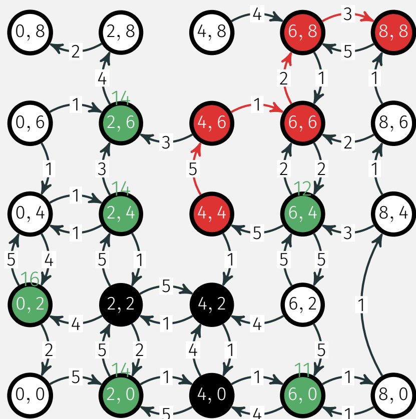


A* heuristique d_2, fin du parcours :

- 8 sommets visités,
- distance obtenue : 11,
- 3 sommets visités inutilement.

Ex2 : A* converge plus vite vers un moins bon chemin





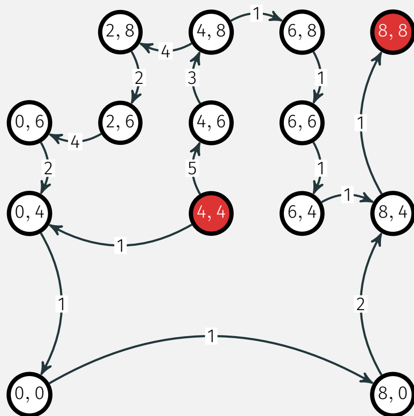
Dijkstra, fin du parcours :

- 14 sommets visités,
- distance obtenue : 7,
- 6 sommets visités inutilement.

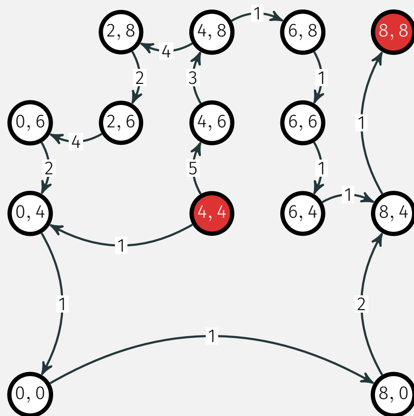
A* heuristique d_2, fin du parcours :

- 8 sommets visités,
- distance obtenue : 11,
- 3 sommets visités inutilement.

Ex3 : A* converge moins vite vers un moins bon chemin...

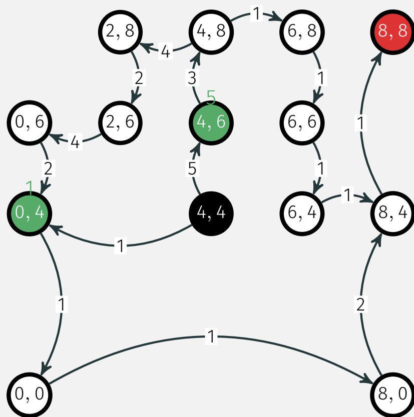


Dijkstra, initialisation

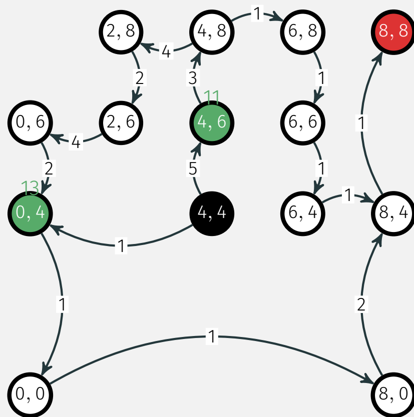


A* heuristique d_1, initialisation

Ex3 : A* converge moins vite vers un moins bon chemin...

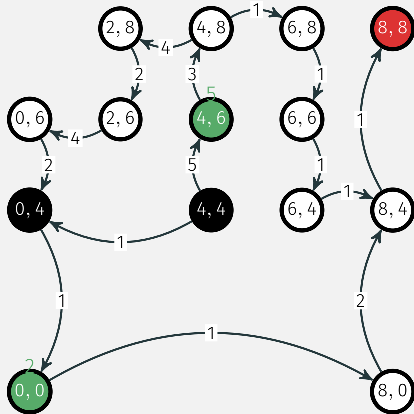


Dijkstra, étape 1

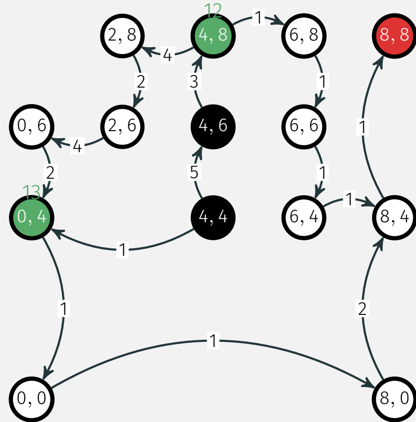


A* heuristique d_1, étape 1

Ex3 : A* converge moins vite vers un moins bon chemin...

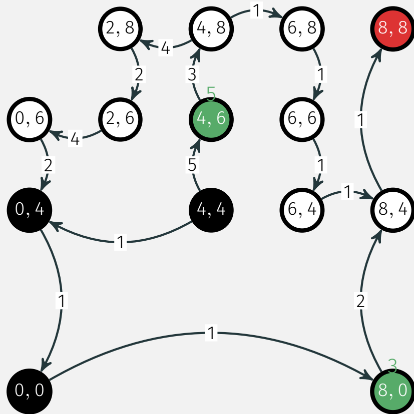


Dijkstra, étape 2

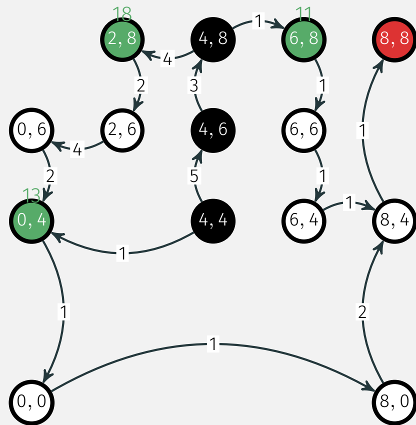


A* heuristique d_1, étape 2

Ex3 : A* converge moins vite vers un moins bon chemin...

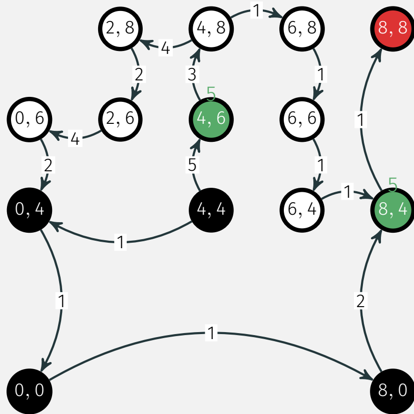


Dijkstra, étape 3

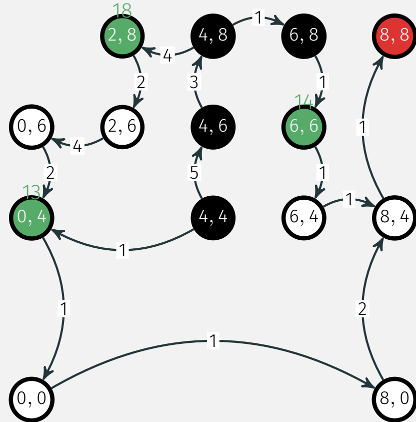


A* heuristique d_1, étape 3

Ex3 : A* converge moins vite vers un moins bon chemin...

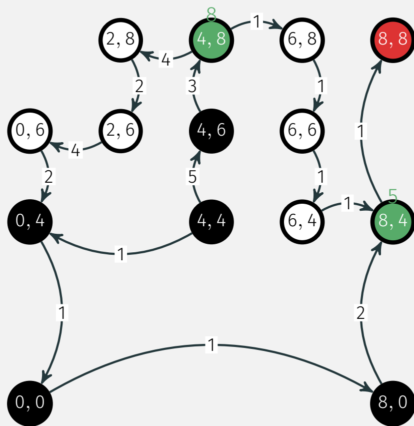


DIJKSTRA, étape 4

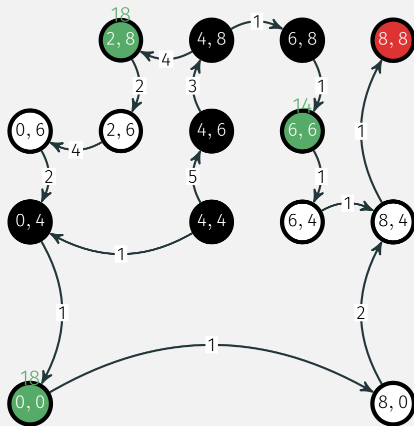


A* heuristique d_1, étape 4

Ex3 : A* converge moins vite vers un moins bon chemin...

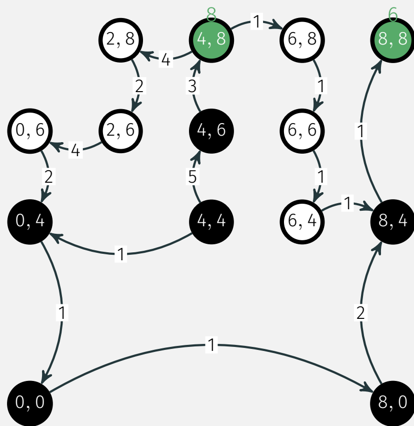


Dijkstra, étape 5

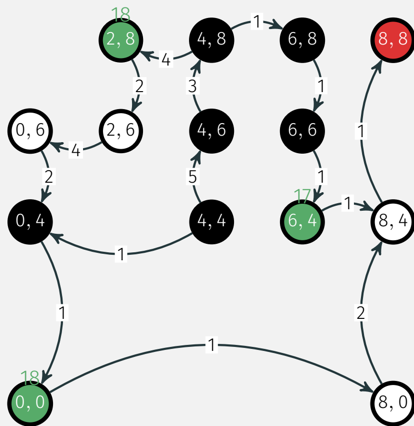


A* heuristique d_1, étape 5

Ex3 : A* converge moins vite vers un moins bon chemin...

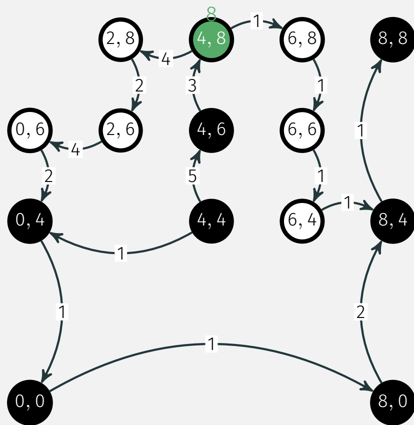


Dijkstra, étape 6

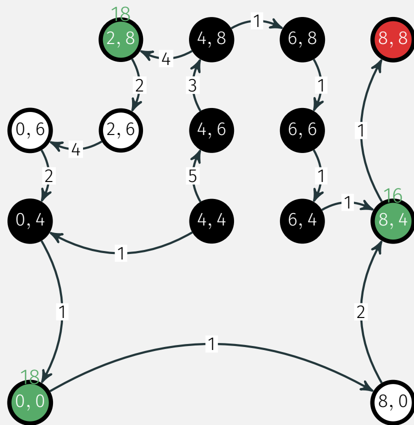


A* heuristique d_1, étape 6

Ex3 : A* converge moins vite vers un moins bon chemin...

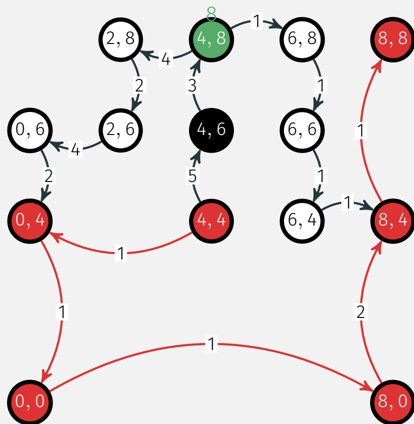


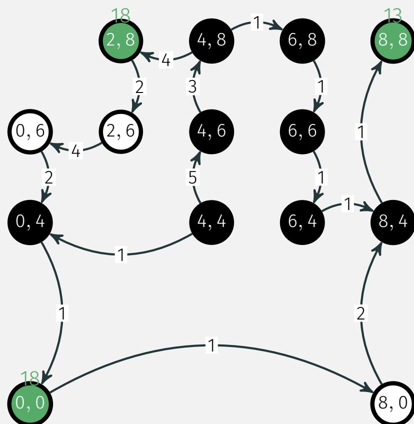
Dijkstra, étape 7



A* heuristique d_1, étape 7

Ex3 : A* converge moins vite vers un moins bon chemin...

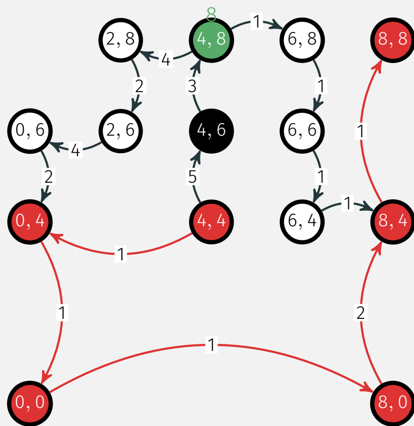


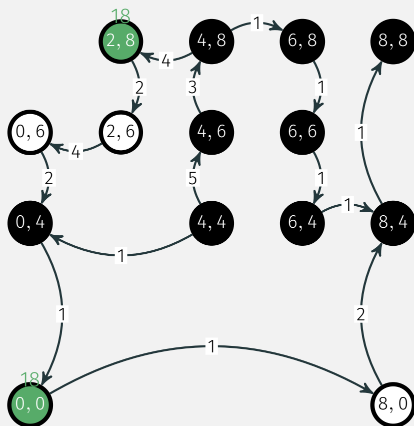


DIJKSTRA, fin du parcours :

- 7 sommets visités,
- distance obtenue : 6,
- 1 sommets visités inutilement.

A* heuristique d_1, étape 8

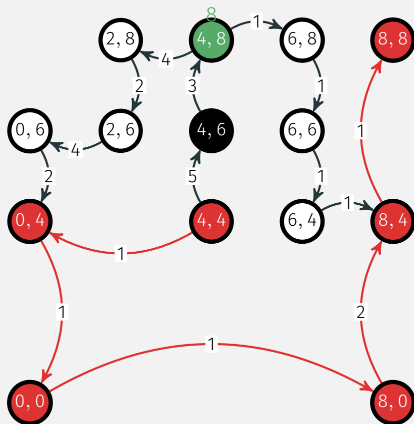


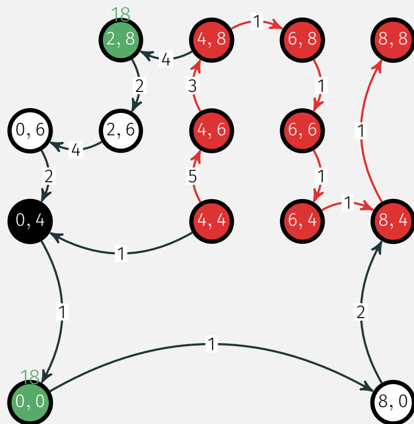


DIJKSTRA, fin du parcours :

- 7 sommets visités,
- distance obtenue : 6,
- 1 sommets visités inutilement.

A* heuristique d_1, étape 9



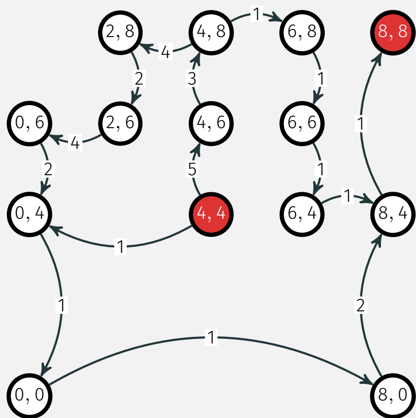


Dijkstra, fin du parcours :

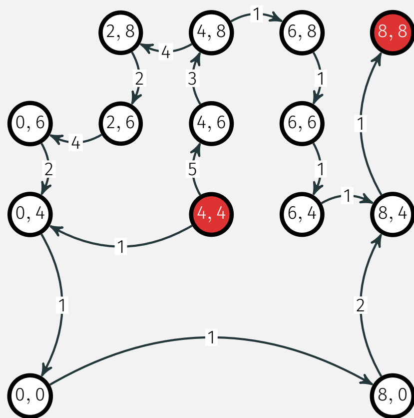
- 7 sommets visités,
- distance obtenue : 6,
- 1 sommets visités inutilement.

A* heuristique d_1, fin du parcours :

- 9 sommets visités,
- distance obtenue : 13,
- 1 sommets visités inutilement.

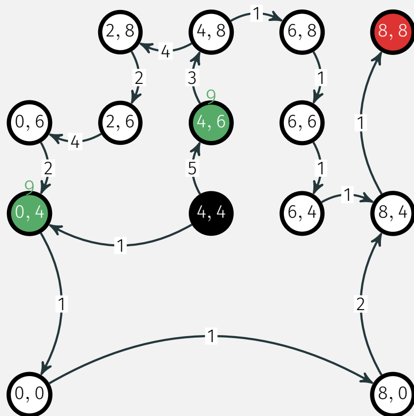


A* heuristique d_inf, initialisation

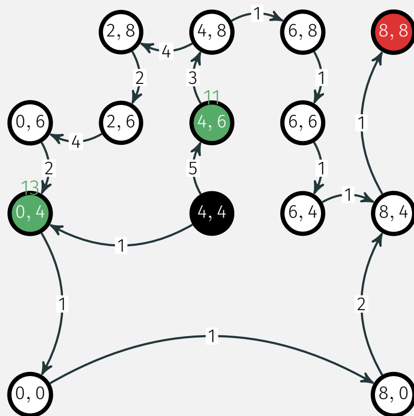


A* heuristique d_1, initialisation

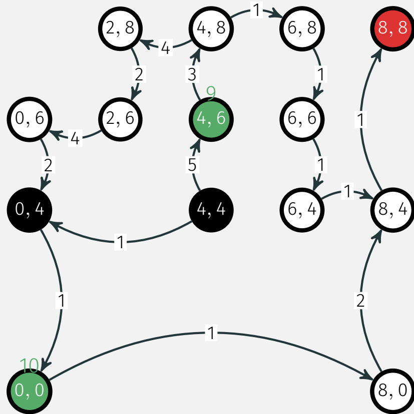
Ex4:...sauf si on change d'heuristique



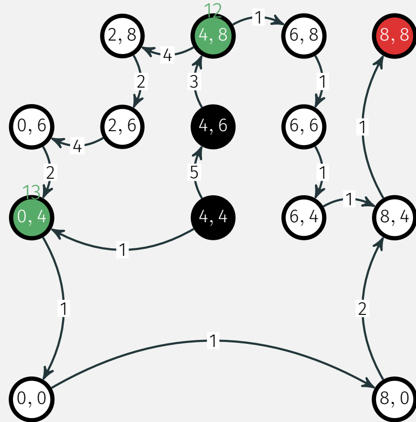
A* heuristique d_inf, étape 1



A* heuristique d_1, étape 1

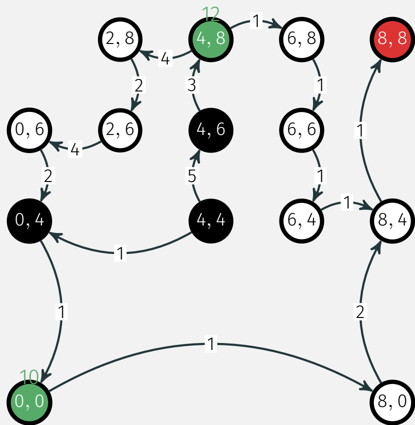


A* heuristique d_inf, étape 2

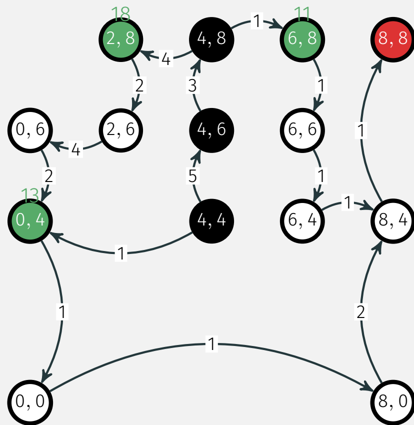


A* heuristique d_1, étape 2

Ex4:...sauf si on change d'heuristique

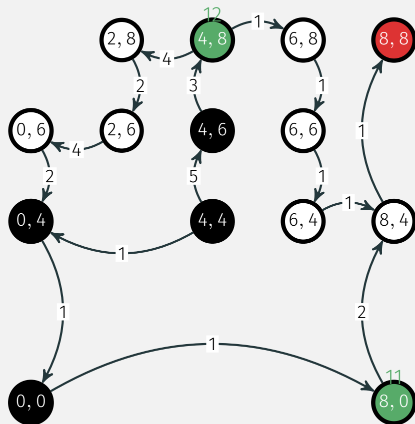


A* heuristique d_inf, étape 3

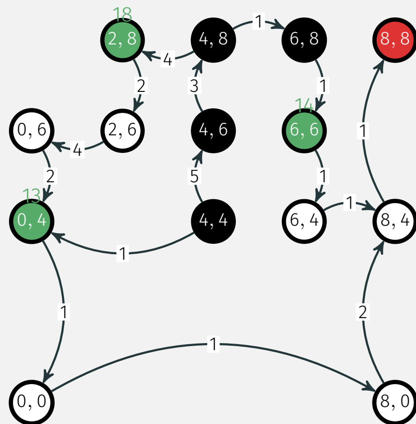


A* heuristique d_1, étape 3

Ex4:...sauf si on change d'heuristique

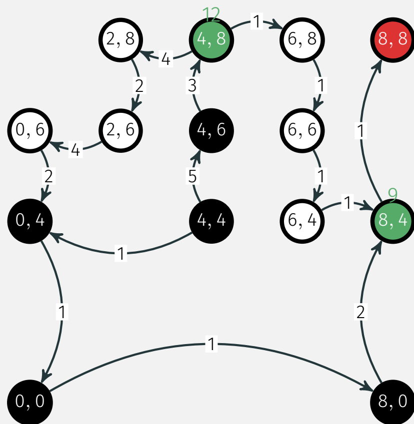


A* heuristique d_inf, étape 4

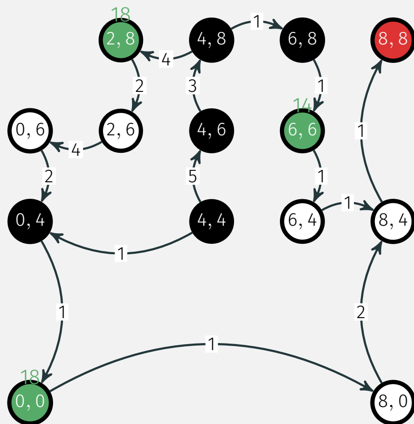


A* heuristique d_1, étape 4

Ex4:...sauf si on change d'heuristique

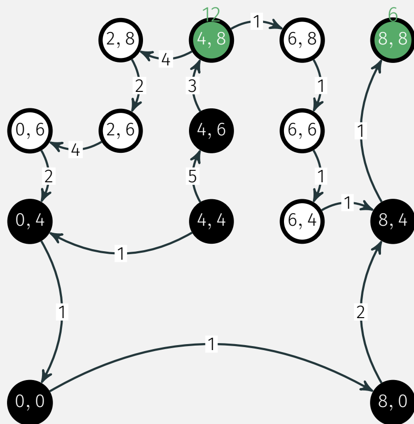


A* heuristique d_inf, étape 5

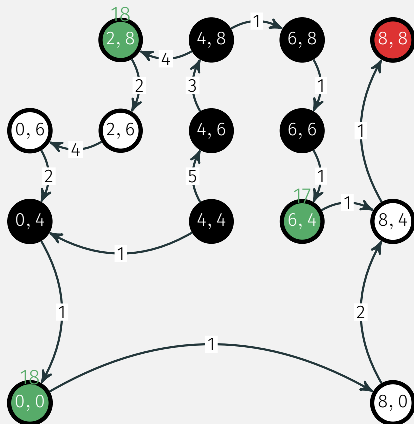


A* heuristique d_1, étape 5

Ex4:...sauf si on change d'heuristique

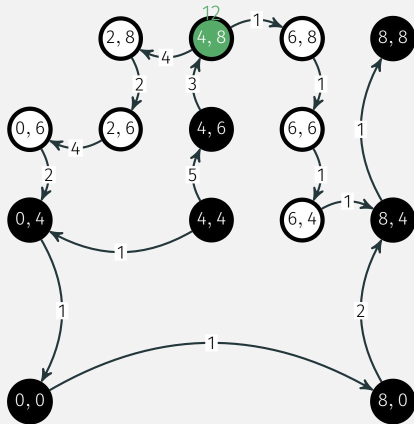


A* heuristique d_inf, étape 6

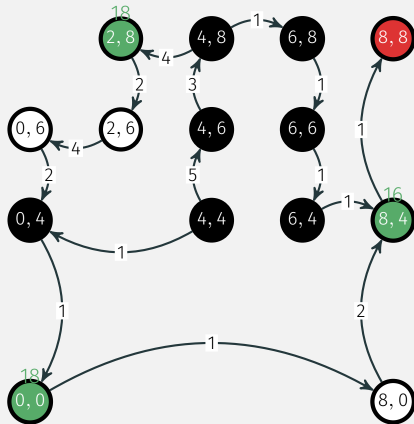


A* heuristique d_1, étape 6

Ex4:...sauf si on change d'heuristique



A* heuristique d_inf, étape 7



A* heuristique d_1, étape 7

Ex4:...sauf si on change d'heuristique

A* heuristique d_inf, fin du parcours :

- 7 sommets visités,
- distance obtenue : 6,
- 1 sommets visités inutilement.

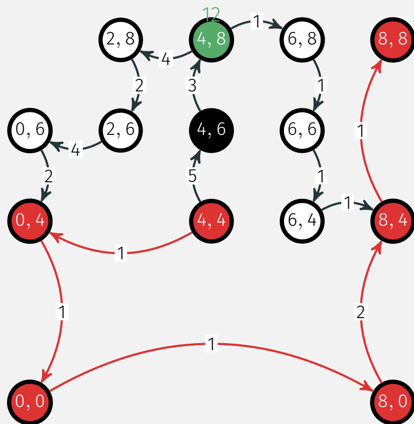
A* heuristique d_1, étape 8

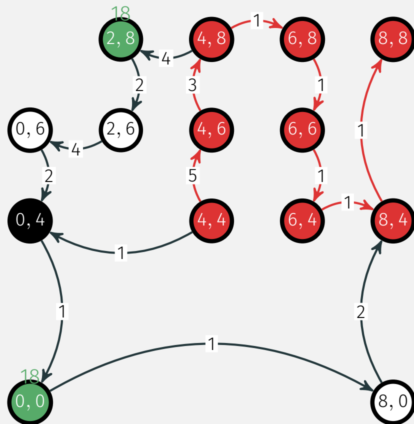
Ex4:...sauf si on change d'heuristique

A* heuristique d_inf, fin du parcours :

- 7 sommets visités,
- distance obtenue : 6,
- 1 sommets visités inutilement.

A* heuristique d_1, étape 9





A* heuristique d_inf, fin du parcours :

- 7 sommets visités,
- distance obtenue : 6,
- 1 sommets visités inutilement.

A* heuristique d_1, fin du parcours :

- 9 sommets visités,
- distance obtenue : 13,
- 1 sommets visités inutilement.