Chapitre (S2) 5 Parcours de graphes pondérés & Applications

Objectifs

- Connaitre l'algorithme de DIJKSTRA permettant de déterminer un plus court chemin dans un graphe pondéré.
- Comprendre l'intérêt d'introduire dans certains cas un terme d'heuristique dans DIJKSTRA, et sa conséquence ; l'algorithme A*.

L'objet de ce chapitre est de présenter des algorithmes de recherche d'un plus court chemin dans des graphes pondérés, orientés ou non. La notion de parcours de graphes est étroitement liée à cette recherche. Nous avons vu dans le Chapitre (S2) 4 que le *parcours en largeur* répond partiellement à notre objectif. *Partiellement* car il ne permet la détermination d'un plus court chemin que dans un graphe non pondéré.

Pour des graphes pondérés, des algorithmes plus efficaces lui sont préférés. L'*algorithme de* DIJKSTRA est l'un d'eux, qui ne s'appliquent qu'à des graphes à *poids positifs* pour des raisons qui seront présentées dans une deuxième partie. Enfin, d'autres algorithmes tirent profit de propriétés particulières des graphes pour proposer des solutions parfois aussi efficaces mais pas toujours optimales. Elles introduisent le concept d'*heuristique* dont une découverte est proposée dans la dernière partie, à travers la présentation de l'*algorithme A**.

Cadre

Dans ce chapitre, tous les graphes sont codés par un *dictionnaire*.

1 GÉNÉRALITÉS

Considérons à présent un *graphe pondéré*, c'est-à-dire dont les arêtes portent une information numérique. Par exemple le graphe ci-dessous.

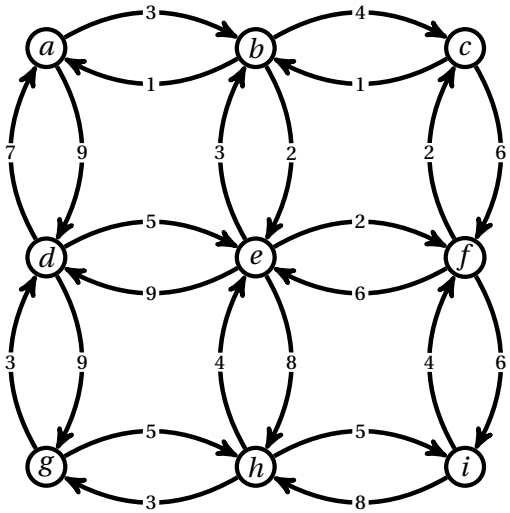


FIGURE 1 : Graphe G_1 orienté et pondéré.

Les *poids* (ou *pondérations*) des arcs sont ici des nombres entiers positifs. Des pondérations négatives et non entières sont possibles. Mais conformément au programme, seuls les graphes pondérés avec des poids positifs sont étudiés. Un paragraphe justifiera ce choix. Dans la suite, le poids d'une arête[1] (u, v) est notée $p(u, v)$. Par exemple, dans le graphe précédent, on a $p(e, d) = 9, p(d, e) = 5$.

1. Ou d'un arc dans le cas d'un graphe orienté.

1.1 Distance (minimale) entre deux sommets

Définition 1

- La *longueur* ou *poids* d'un chemin dans un graphe pondéré est désormais définie comme la somme des poids de ses arêtes.
- La définition de la *distance* entre deux sommets n'est pas modifiée : c'est *la longueur d'un*[a] *plus court chemin* entre les deux sommets.
$$\mathrm{d}_u[v] = \min_{\gamma \in \Gamma_u(v)} \delta(\gamma).$$
- Un *plus court chemin* entre deux sommets u et v d'un graphe est un chemin d'extrémités u et v dont la longueur est égale à sa distance $\mathrm{d}_u[v]$.

Exemple 1 Sur le graphe G_1, le chemin $\gamma = (e, d, g, h, i)$ est de longueur :
$$\delta(\gamma) = p(e,d) + p(d,g) + p(g,h) + p(h,i) = 9 + 9 + 5 + 5 = 28.$$

1.2 Une adaptation de BFS ?

Dans un graphe non pondéré, le parcours en largeur permet le calcul des distances entre deux sommets. On peut toutefois interpréter ce calcul comme celui des distances dans un graphe pondéré dont tous les poids valent 1. Se pose alors la question d'adapter le parcours pour calculer les distances dans un graphe pondéré. Mais cette idée se heurte à la nature même du parcours qui visite les sommets voisins d'un sommet, sans se soucier de la valeur du poids de l'arête. Ce qui interdit toute mise en oeuvre directe du parcours en largeur pour calculer les distances.

Toutefois, pour contourner la difficulté précédente, on pourrait transformer tout graphe pondéré avec des *poids entiers positifs* par un graphe dont tous les poids des arêtes seraient 1. Sur chaque arête de poids p, $(p-1)$ sommets seraient ajoutés, toutes les arêtes portant désormais un poids 1. Le parcours en largeur deviendrait alors pertinent. Cette solution, bien que séduisante, présente plusieurs inconvénients. Tout d'abord, ajouter *artificiellement* des sommets rend les parcours moins efficaces puisque leur complexité dépend directement du nombre de sommets. Avec des poids dont les valeurs seraient déraisonnablement élevées, on imagine facilement la faible efficacité de la solution. De plus, le parcours passerait l'essentiel de son temps à visiter des sommets qui, dans le graphe initial, n'existent pas et, finalement, ne présente que peu d'intérêt.

a. Noter l'usage du déterminant indéfini *un*. Des chemins différents peuvent relier deux sommets, certains d'entre eux étant de même distance entre les deux sommets.

2 PARCOURS : L'ALGORITHME DE DIJKSTRA

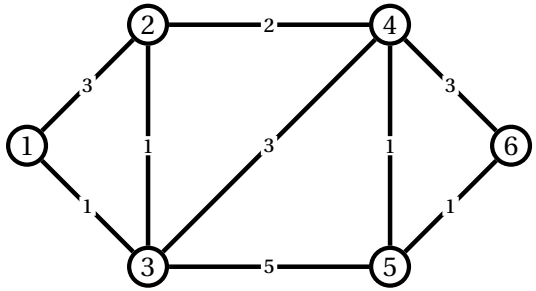
Edsger Dijkstra (1930 - 2002) est un informaticien néerlendais qui a reçu le prix Turing en 1972 pour ses nombreuses contributions majeures. Adepte des beaux algorithmes, il a largement contribué au développement de la science et de l'art des langages de programmation. De nombreux aphorismes lui sont attribués, notamment : *La recherche d'un plus court chemin d'un graphe n'est jamais celui que l'on croit ; il peut surgir de nulle part et la plupart du temps, il n'existe pas* ; ou encore : *Les progrès ne seront possibles que si nous pouvons réfléchir sur les programmes sans les imaginer comme des morceaux de code exécutable.* En 2002, en son honneur, le prix PoDC est renommé prix Dijkstra, qui récompense des travaux importants dans le domaine des système distribués.

Nous commençons par détailler l'algorithme sur un premier exemple, avant de formaliser un peu plus, puis de le coder en Python. Nous reprenons le vocabulaire adopté dans le précédent chapitre : un sommet est soit *non découvert*, soit *découvert*, soit *visité*.

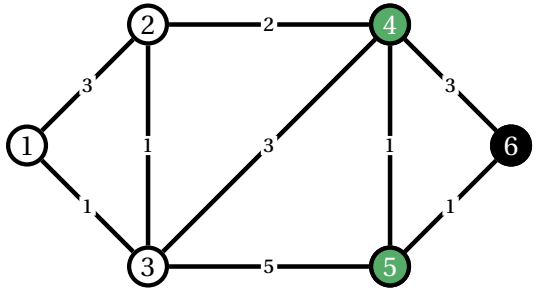
2.1 Exemple

Commençons par détailler l'algorithme sur un premier exemple, à partir par exemple du sommet 6 ci-dessous. En fin d'algorithme, nous obtiendrons les distances (minimales) depuis 6 **jusqu'à n'importe quel sommet du graphe** ; elles seront stockées ultérieurement dans un dictionnaire, pour le moment nous les indiquons dans un tableau. Voici les grandes étapes :

- on initialise un tableau des distances avec 0 (pour le sommet source), et $+\infty$ pour les autres. On marque le sommet source (6 ici) comme visité.
- On explore les voisins non visités v de $u = 6$: pour chaque tel sommet v, on indique alors dans le tableau la plus petite valeur entre l'ancienne distance, et celle ayant permis de découvrir v (c'est-à-dire ici $0 + p(u,v)$ où $p(u,v)$ désigne le poids de l'arc $u \to v$).
- On recommence le procédé avec u qui est le sommet de distance minimale dans le tableau, et tant qu'il reste des sommets découverts non visités.

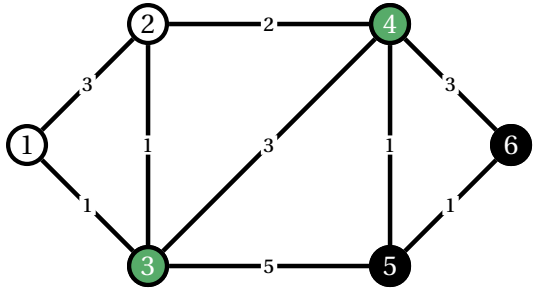


Sommet	1	2	3	4	5	6
6	+∞	+∞	+∞	+∞	+∞	0



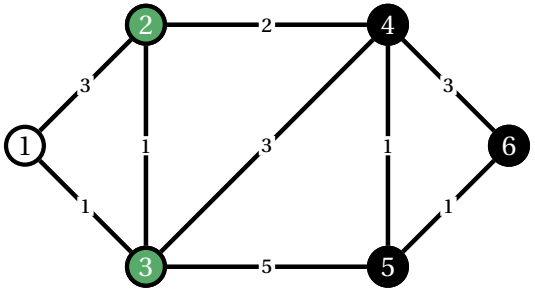
Sommet	1	2	3	4	5	6
6	+∞	+∞	+∞	3	1	

On recommence ensuite avec le sommet 5 qui est celui de distance minimale. Le sommet 6 est marqué comme visité : sa colonne de distances ne doit plus changer.



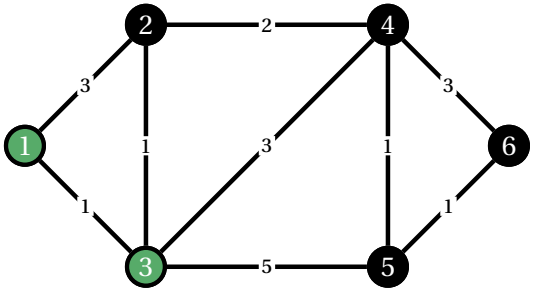
Sommet	1	2	3	4	5	6
6	+∞	+∞	+∞	3	1	0
5	+∞	+∞	6	2		

On recommence ensuite avec le sommet 4 qui est celui de distance minimale. On le marque comme visité.



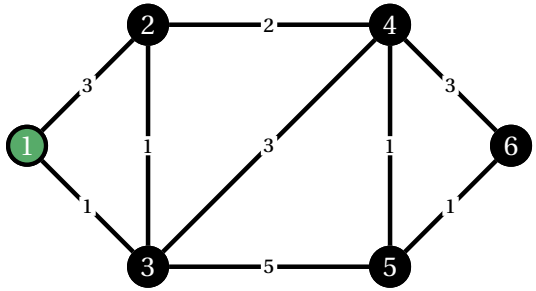
Sommet	1	2	3	4	5	6
6	+∞	+∞	+∞	3	1	0
5	+∞	+∞	6	2		
4	+∞	4	5			

On recommence ensuite avec le sommet 2 qui est celui de distance minimale. On le marque comme visité.



Sommet	1	2	3	4	5	6
6	+∞	+∞	+∞	3	1	0
5	+∞	+∞	6	2		
4	+∞	4	5			
2	7		5			

On recommence ensuite avec le sommet 3 qui est celui de distance minimale. On le marque comme visité.



Sommet	1	2	3	4	5	6
6	+∞	+∞	+∞	3	1	0
5	+∞	+∞	6	2		
4	+∞	4	5			
2	7		5			
2	6					

On marque ensuite 1 comme visité : il n'y a plus aucun sommet découvert et non visité. L'algorithme est terminé.

BILAN. Dans chaque colonne du tableau, la dernière valeur contient alors la distance minimale.

Sommet	1	2	3	4	5	6
Distance min (à 6)	6	4	5	2	1	0

Remarque 1 Au moment de la sélection du prochain sommet à visiter, si plusieurs sont de distance minimale dans le tableau, on peut choisir l'un d'entre eux arbitrairement.

Exercice 1 [Sol 1] Sur le graphe précédent, appliquer l'algorithme de DIJKSTRA à partir du sommet 2.

2.2 Principe de sous-optimalité

L'*algorithme de* DIJKSTRA repose sur le *principe de sous-optimalité* adapté à la recherche d'un plus court chemin. La proposition qui suit est une des étapes de la preuve de l'algorithme de DIJKSTRA, qui est difficile.

Proposition 1 | Principe de sous-optimalité
Soit c un *plus court chemin* allant d'un sommet u vers un sommet v d'un graphe. Notons $u \overset{c}{\rightsquigarrow} v$ un tel plus court chemin.
Alors si c passe par un sommet intermédiaire s, $u \overset{c_1}{\rightsquigarrow} s$ et $s \overset{c_2}{\rightsquigarrow} v$ sont aussi des plus courts chemins.

Ce résultat affirme que l'optimalité de la solution du *problème* du calcul d'un plus court chemin passe par l'optimalité des solutions des *sous-problèmes* de calculs de plus courts chemins.

Dit autrement, 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. De tels problèmes peuvent être résolus par des méthodes dites de *programmation dynamique*[2].

Bien qu'elle puisse être omise en première lecture, la démonstration de ce résultat se fait par l'absurde.

Preuve Si G = (S, A) est un graphe pondéré de valuation définie par une fonction p, chaque arc $(v_i, v_j) \in$ S a un poids $p(v_i, v_j)$. Pour tout chemin $\gamma = (x_0, x_1, \dots, x_k)$ dans G, le poids du chemin est : $\delta(\gamma) = \sum_{i=0}^{k-1} p(x_i, x_{i+1})$. Considérons un *plus court chemin* c du sommet u au sommet v, il vérifie alors $d_u[v] = \delta(c)$. Soit s un sommet intermédiaire de ce plus court chemin, de sorte que $u \overset{c_1}{\rightsquigarrow} s \overset{c_2}{\rightsquigarrow} v$.
Supposons par l'absurde qu'il existe un chemin c_1' plus court pour aller de u à s : $\delta(c_1') < \delta(c_1)$. Alors il existe un chemin $u \overset{c_1'}{\rightsquigarrow} s \overset{c_2}{\rightsquigarrow} v$ de u à v de poids : $\delta(c_1') + \delta(c_2) < \delta(c_1) + \delta(c_2) = d_u[v]$, ce qui est absurde car $s \overset{c_1}{\rightsquigarrow} u \overset{c_2}{\rightsquigarrow} t$ est un *plus court* chemin. — **Contradiction**
Le chemin c_1 est donc un plus court chemin de u à s : $d_u[s] = \delta(c_1)$. La même analyse vaut pour c_2.

Cette proposition permet donc d'expliquer le choix du sommet d'étiquette minimale à chaque étape : en effet, celui qui est actuellement d'étiquette minimale ne pourra jamais plus voir son étiquette améliorée par un autre choix de sommet (car les autres sommets non encore visités sont d'étiquette supérieure).

2.3 Présentation générale

L'*algorithme de* DIJKSTRA calcule *tous les plus courts chemins entre un sommet donné et les autres sommets d'un graphe.* Sa mise en oeuvre suit le principe de sous-optimalité en déterminant, 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

2. Ce thème sera abordé en deuxième année.

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 !

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 :

$$d_u[u] = 0, \quad \text{et :} \quad \forall v \in S \setminus \{u\}, \quad d_u[v] = +\infty.$$

Étape 1. L'algorithme part du sommet u et le marque comme *visité*. Les n_1 sommets *voisins non visités* de u, notés $v_{1,1}, \dots, v_{1,n_1}$, sont *découverts* et on procède au *relâchement* de chaque arc $(u, v_{1,i})$, à savoir :

- 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.

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

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

Étape 2. Parmi tous les sommets découverts $\{v_{1,1}, \dots, v_{1,n_1}\}$, 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 effectivement la plus courte distance entre u et v_1. Sa valeur dans le tableau ne doit plus évoluer. Le sommet v_1 sert alors de nouveau sommet de départ et est marqué comme *visité*. Les n_2 sommets voisins *non visités* de v_1, notés $v_{2,1}, \dots, v_{2,n_2}$, sont alors *découverts* et on procède au *relâchement* de 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* 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.

File de priorité. La présentation précédente de l'algorithme de Dijkstra ne précise pas dans quelle structure de données les sommets *découverts* sont stockés. Il en est de même de l'identification du sommet de distance minimale contenu dans cette structure. Or le choix de cette structure a des conséquences sur les performances de l'algorithme. Si l'algorithme de Dijkstra est vu comme une généralisation du parcours en largeur, la file est une solution possible. Mais l'extraction du *bon* sommet est coûteuse sans parler des éventuelles mises à jour des distances associées aux sommets déjà présents dans la structure.

La structure de données la plus adaptée pour répondre aux besoins de l'algorithme est une *file de priorité*. Chaque information stockée dans la structure est accompagnée d'une deuxième information appelée sa *priorité*. L'intérêt essentiel de la structure est de permettre l'*extraction* et l'*ajout* d'information en réorganisant dynamiquement la structure pour préserver l'ordre des priorités avec, pour une *file de priorité* de taille n, une complexité en $O(\log n)$.

Les files de priorités n'étant pas au programme, nous utiliserons un module spécifique qui implémente une telle structure : le module `heapq`[3] préalablement chargé par :

```
import heapq
```

Les fonctions de manipulation de la structure sont les suivantes :

- **[Création d'une file de priorité]** On déclare d'abord une liste vide `hq` puis `heapq.heapify(hq)` transforme `hq` en une file de priorité. Une liste non vide de taille n peut également être transformée en une file de priorité de la même façon. La complexité temporelle de l'opération est $O(n)$.
- **[Ajout]** L'instruction `heapq.heappush(hq,x)` ajoute un élément `x` dans une file de priorité `hq`. Si `hq` comporte initialement n éléments, la réorganisation qui peut en découler est de complexité en $O(\log n)$.
- **[Extraction]** L'instruction `heapq.heappop(hq)` supprime et renvoie l'élément de *priorité minimale* de la file de priorité `hq`. Si la file est vide, un message d'erreur est renvoyé. Là encore, si `hq` comporte initialement n éléments, en raison d'une possible nécessité de réorganiser les données, la complexité de l'opération est en $O(\log n)$.

3. Les files de priorité peuvent être implémentées à l'aide de tableaux ayant la propriété de tas (autre structure de données). En anglais, un tas se dit *heap* et une file de priorité ainsi implémentée est désignée par *heap queue*.

- **[Test file de priorité vide]** Le booléen `len(hq) == 0` (`len(hq) > 0`) ou renvoie **`True`** si la file de priorité `hp` est vide, **`False`** si elle n'est pas vide. L'opération est de complexité $O(1)$.

Exemple : après le remplissage d'une liste avec des entiers pris au hasard dans l'intervalle $[25, 75[$ (ligne 7), puis sa transformation en file de priorité (ligne 11), une boucle (lignes 15 à 17) vide la file de priorité et affiche les éléments dans l'ordre de leur extraction. La colonne de droite ci-dessous présente un exemple d'exécution du code.

Noter que dans cette implémentation, la priorité minimale de l'élément situé en tête de file[4].

```
>>> 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
[54, 73, 40, 45, 47, 37, 57, 74, 66, 35]
>>> heapq.heapify(hq) # Définition d'une file de priorité hq
>>> hq
[35, 45, 37, 54, 47, 40, 57, 74, 66, 73]
>>> while len(hq) > 0: # # Extraction des entiers de hq
...     x = heapq.heappop(hq)
...     print(x)
...
35
37
40
45
47
54
57
66
73
74
```

4. La réorganisation des données lors de chaque ajout ou extraction n'est pas évidente à comprendre sauf à voir l'organisation sous forme arborescente. Ce sujet n'étant pas au programme, tout lecteur curieux est invité à consulter un ouvrage spécilisé, comme le livre de *Cormen, Leiserson, Rivest - Introduction à l'algorithmique*, à votre disposition au CDI.

L'intérêt de cette structure pour l'algorithme de DIJKSTRA est de permettre l'identification des sommets de distances minimales ; dans l'exemple précédent, chaque sommet a la même priorité (la liste ne comporte pas de couple). Désormais, on souhaite que chaque fois qu'un sommet est découvert, un couple d'informations soit stocké dans une file de priorité. Le premier élément du couple est la valeur de la distance calculée après relâchement d'une arête (v_i, v_j). Le second élément du couple est l'étiquette v_j du sommet découvert. C'est la première information du couple qui permet la réorganisation des données lors d'un enfilement ou d'un défilement.

Le code ci-dessous illustre cette idée.

```
>>> 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')]
>>>
>>> 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.

Remarque 2 Une implémentation par listes (comme pour les files et piles) est aussi possible, mais rédhibitoire en terme de complexité ; plus précisément, on pourrait considérer une liste de couples où dans chaque couple on indiquerait le sommet ainsi que son poids. Le soucis majeur de cette représentation consiste

en l'étape de recherche du sommet de poids minimal (complexité linéaire, alors que le module présenté ci-dessous donnera une complexité logarithmique).

MISE EN OEUVRE PYTHON. Il est à présent possible de mettre en oeuvre l'algorithme de DIJKSTRA dans le langage Python. Le code est similaire à celui de parcours en largeur. La file est remplacée par une file de priorité qui contient des couples (distance, sommet) comme indiqué ci-dessus. 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 clé, 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`.

```
def dijkstra(g, v_init):
    visited = {x : False for x in g}              dico des sommets visités
    pred = {x : None for x in g}                  dico des predecesseurs
    dist = {x : float('inf') for x in g}          dico des distances
    dist[v_init] = 0                              vinit est à distance 0 de lui-même
    hq = [(0, v_init)]
    heapq.heapify(hq)                             création de la FP
    while len(hq) > 0:                            visite des sommets
        dv, v = heapq.heappop(hq)                 extraction du sommet de prio min
        if not visited[v]:
            visited[v] = True
            for w, dvw in g[v]:                   parcours des voisins non visités de v
                if not visited[w]:
                    dw = dv + dvw                 relâchement de l'arête (v,w)
                    if dw < dist[w]:
                        dist[w] = dw              maj de la distance min
                        pred[w] = v               maj du prédécesseur
                        heapq.heappush(hq, \
                            (dw, w))              ajout dans la FP
    return dist, pred
```

Ci-dessous un exemple de mise en oeuvre avec le dictionnaire `g1` associé au graphe de la figure Figure 1.

```
>>> g1 = {
...   '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(g1, '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. La complexité du code dépend largement de celle de la file de priorité. Les opérations à l'ajout ou à l'extraction dans une telle file de priorité sont de complexité au pire en $O(\log n)$ où n est la taille de la file.

On peut donc analyser la *complexité temporelle* de `dijkstra` de la façon suivante.

- **[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 clés 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 et chaque visite peut conduire à l'ajout d'un élément dans la file. La file de priorité peut donc contenir jusqu'à $|A|$ éléments. Les opérations d'ajout et d'extraction ayant un coût logarithmique, chaque opération sur la file a donc un coût $O(\log|A|)$. Or $|A| \leqslant |S|^2$ de sorte que $\log|A| = O(\log|S|)$. D'où un coût total $O(|A|\log|S|)$ ou encore en $O(|S|^2 \log|S|)$.

Pour plus de détails, le livre de *Cormen, Leiserson, Rivest - Introduction à l'algorithmique* est disponible au CDI.

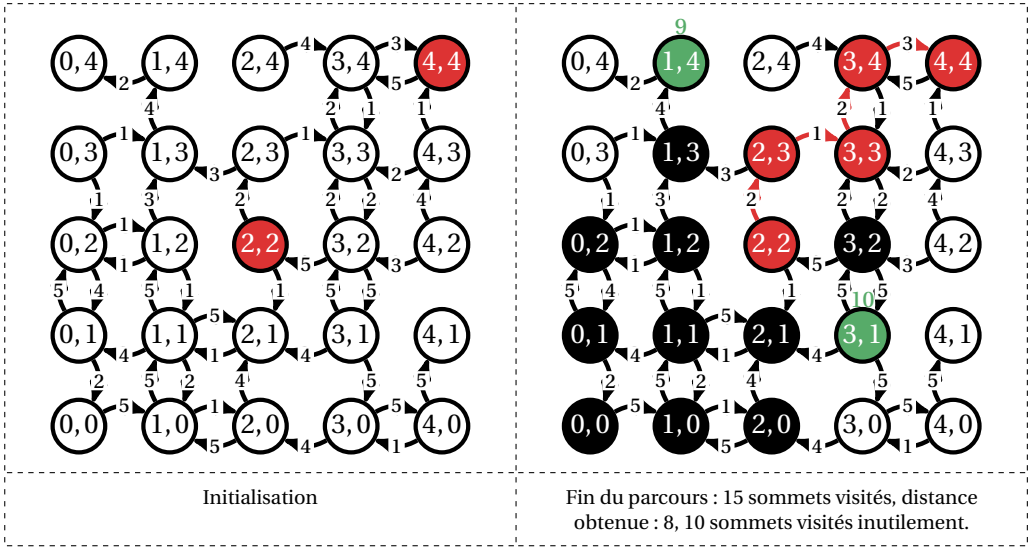
APPLICATION À LA RECHERCHE DE PLUS COURT CHEMIN. La fonction `dijkstra(g,v_init)` peut-être transformée[5] en une fonction

5. C'est l'objet d'un exercice du TP.

`dijkstra_path(g,v_init,v_fin)` prend 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` et `C` est un meilleur chemin, *i.e.* une liste de sommets, menant de `v_init` à `v_fin`.

Notons qu'il n'est pas ici nécessaire de parcourir tout le graphe : on peut s'arrêter dès que l'on trouve `v_fin`. Ci-dessous l'illustration de la recherche d'un plus court chemin du sommet $(2,2)$ au sommet $(4,4)$:



Initialisation

Fin du parcours : 15 sommets visités, distance obtenue : 8, 10 sommets visités inutilement.

Notons que 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. Pour l'améliorer, il faudrait "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` et augmenter celle 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*.

3 ALGORITHME A*

3.1 Informer l'algorithme de DIJKSTRA

L'algorithme de DIJKSTRA peut être utilisé pour déterminer un plus court chemin entre un sommet de départ `v_init` et un sommet d'arrivée `v_fin`. Il suffit pour cela soit de lui faire rechercher tous les plus courts chemins issus de `v_init`, soit d'arrêter la recherche dès que le sommet `v_fin` a été visité. Dans les deux cas, pour construire la solution, l'algorithme explore un grand nombre de sommets dont certains ne semblent pas toujours pertinents au regard du résultat escompté.

Pour illustrer cette idée, considérons la carte de France (Figure 2) en vue de trouver un plus court itinéraire routier de Bordeaux à Strasbourg. Un rapide coup d'oeil nous permet généralement d'identifier les principaux axes routiers utiles. En particulier, sont immédiatemment rejetés tous les itinéraires qui auraient tendance à augmenter la distance à parcourir. Mais comment un algorithme peut-il faire de même ? La réponse est simple : il ne peut pas. Tout au moins, pas si on ne lui apporte pas d'information complémentaire. L'algorithme de DIJKSTRA est un exemple d'*algorithme non informé*. De fait, pour trouver un itinéraire optimal, il va explorer un très grand nombre d'itinéraires parmi lesquels certains ne présenteront aucun intérêt pour répondre à notre besoin. Alors, comment l'informer et l'orienter dans sa recherche ?

Notre lecture de la carte nous mène à orienter nos recherche d'itinéraire dans la *direction* Bordeaux-Strasbourg. Les directions Bordeaux-Nantes, Bordeaux-Toulouse, Bordeaux-Marseille sont d'emblée éliminées du champ d'investigation. Cette *attitude* peut être partiellement traduite sous forme algorithmique en orientant les recherches : on parle d'*algorithme informé*. L'algorithme A* [6] appartient à cette catégorie. Dans une certaine mesure, il est une généralisation de l'algorithme de DIJKSTRA qui peut trouver les mêmes solutions optimales que ce dernier, sous réserve que certaines conditions soient satisfaites [7].

3.2 Principe de A*

L'idée générale de l'*algorithme* A* est de favoriser les chemins qui mènent plus rapidement vers la solution. Bien évidemment, il n'existe pas un seul moyen de répondre à cet impératif car sinon, cela signifierait qu'on a trouvé *la* solution optimale qui est justement ce que l'on cherche. Mais on peut *orienter* les choix de l'algorithme en

6. Prononcer A *étoile* en français, A *star* en anglais.
7. Il n'entre pas dans le cadre de ce cours de développer ce point.

FIGURE 2 : Carte de France et de ses principaux axes routiers.

modifiant dans le code de DIJKSTRA la priorité p_w des sommets w : elle n'est plus simplement sa distance depuis `v_init` mais sa distance depuis `v_init` à laquelle on ajoute une estimation $h(w, v_{fin})$ du coût de ce qu'il reste à parcourir de w jusqu'à `v_fin` : $\boxed{p_w = \mathrm{d}_{v_{\text{init}}}[w] + \mathbf{h(w, v_{fin})}}$.

La fonction h est appelée une *heuristique*, elle permet d'estimer la proximité de deux sommets : $h(v, w)$ est le *coût estimé* du chemin le moins coûteux de v à w. Plusieurs heuristique sont possibles, l'efficacité de l'algorithme A* étant conditionnée au choix d'une heuristique adaptée à la situation.

On trouvera ci-après le code de la fonction `a_star_path(g, v_init,v_fin, h)` 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}                    dico des sommets visités
    pred = {x : None for x in g}                        dico des prédécesseurs
    dist = {x : float('inf') for x in g}                dico des dist
    dist[v_init] = 0
    hq, N = [(h[v_init], v_init)], 0                    FP, compteur des sommets vis.
    heapq.heapify(hq)
    while len(hq) > 0 and not visited[v_fin]:
        pv, v = heapq.heappop(hq)                       extraction du sommet de prio min
        if not visited[v]:
            visited[v], N = True, N+1                   maj du compteur
            for w, dvw in g[v]:                         parcours des vois. non visités de 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        maj de la dist et du pred
                        heapq.heappush(hq, (pw, w))     stockage dans la
FP
    # Calcul d'un chemin _____________
    if not visited[v_fin]:                              cas où vfin n'est pas accessible
        print("Pas de chemin de "+str(v_init)+" à "+str(v_fin))
    else:                                               construction du chemin
        C = [v_fin]
        while C[0] != v_init:
            w = pred[C[0]]
            C = [w] + C
        return N, dist[v_fin], C
```

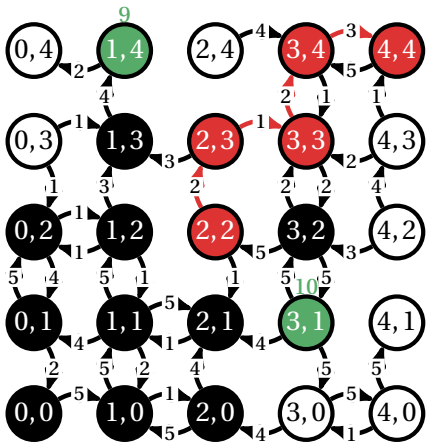
Le code étant très proche de celui de DIJKSTRA, le coût de toutes les opérations y est le même. Seule s'ajoute la prise en compte de l'heuristique dont le coût peut, dans notre cas, être pris comme constant. La complexité de l'algorithme A* est donc toujours en $O(|A|)\log|S|)$ soit en $O\big(|S|^2\log|S|\big)$.

3.3 Comparaisons DIJKSTRA/A*

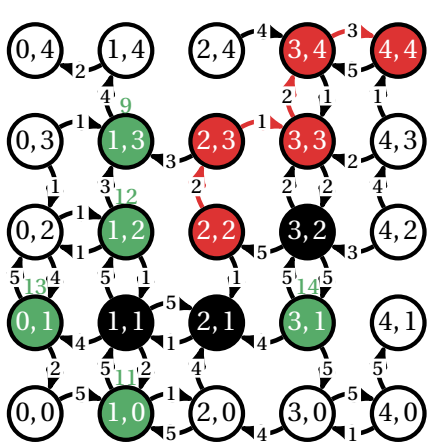
Illustrons le déroulement de DIJKSTRA et de A* sur différents graphes et différentes heuristiques. L'étiquette d'un sommet sera ses coordonnées dans le plan et nous utiliserons comme heuristiques les distances suivantes de $\mathbb{R}^2$: soient $u(x_1,y_1)$, $v(x_2,y_2)$ et $p\in[1,+\infty]$, on pose : $\begin{cases} d_p(u,v) &= (|x_2-x_1|^p+|y_2-y_1|^p)^{\frac{1}{p}} \\ d_\infty(u,v) &= \max(|x_2-x_1|,|y_2-y_1|)\,. \end{cases}$

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*).

Figure 3 – EXEMPLE 1 : A* CONVERGE PLUS VITE VERS UN AUSSI BON CHEMIN

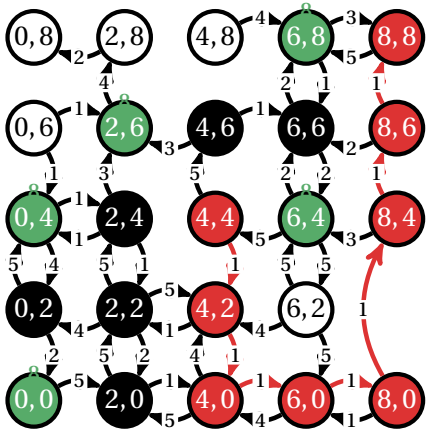


Dijkstra : 15 sommets visités, distance obtenue : 8, 10 sommets visités inutilement.

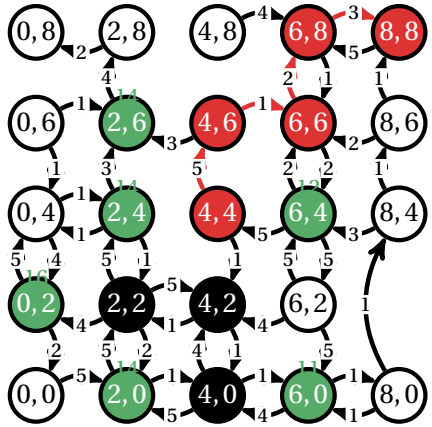


A* heuristique d_1 : 8 sommets visités, distance obtenue : 8, 3 sommets visités inutilement.

Figure 4 – EXEMPLE 2 : A* CONVERGE PLUS VITE VERS UN MOINS BON CHEMIN

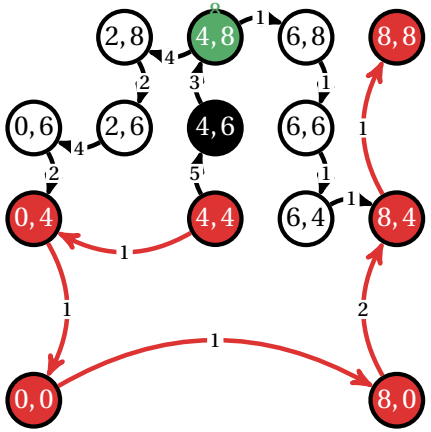


Dijkstra : 14 sommets visités, distance obtenue : 7, 6 sommets visités inutilement.

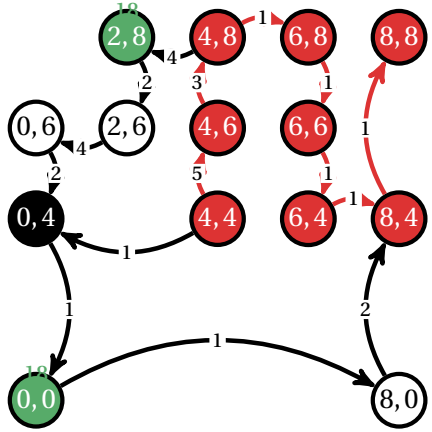


A* heuristique d_1 : 8 sommets visités, distance obtenue : 11, 3 sommets visités inutilement.

Figure 5 – EXEMPLE 3 : A* CONVERGE MOINS VITE VERS UN MOINS BON CHEMIN...

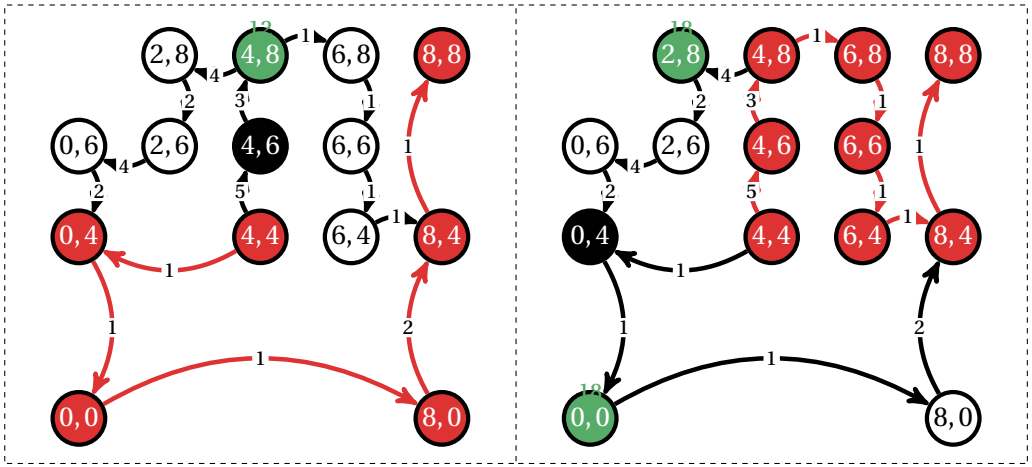


Dijkstra : 7 sommets visités, distance obtenue : 6, 1 sommet visité inutilement.



A* heuristique d_1 : 9 sommets visités, distance obtenue : 13, 1 sommet visité inutilement.

Figure 6 – EXEMPLE 4 : ...SAUF SI ON CHANGE D'HEURISTIQUE



A* heuristique d_∞ : 7 sommets visités, distance obtenue : 6, 1 sommet visité inutilement.

A* heuristique d_1 : 9 sommets visités, distance obtenue : 13, 1 sommet visité inutilement.

3.4 Applications

L'algorithme A* présente l'avantage sur celui de DIJKSTRA de réduire considérablement l'*exploration* d'un graphe, avantage qui permet souvent de trouver une solution plus rapidement, surtout quand les graphes ont des tailles importantes. Cet avantage est d'autant plus important que certains graphes présentent un nombre très élevés de sommets, intertisant parfois leur définition préalable. On doit alors se tourner vers une exploration du graphe qui découvre, au fur et à mesure qu'il les construit, ses sommets lors du parcours.

C'est le cas des graphes où chaque sommet peut être associé à la configuration d'un jeu comme le taquin[8], l'âne rouge[9] et le Rush-Hour[10]. Chaque déplacement d'une pièce dans ces jeux peut définir le sommet d'un graphe. Chaque arête entre deux sommets n'existe que si le passage d'une configuration à une autre du jeu est autorisée. Résoudre le jeu revient alors *simplement* à trouver un plus court chemin entre une configuration initiale et la configuration finale du jeu, généralement connue! Mais la difficulté est clairement la définition du graphe. Le nombre de configurations, parfois extrêmement élevé, ne permet pas sa construction exhaustive. Et même si c'était le cas, le parcours de ce dernier avec l'algorithme de DIJKSTRA ne serait pas raisonnable.

8. https://fr.wikipedia.org/wiki/Taquin
9. https://en.wikipedia.org/wiki/Klotski
10. https://en.wikipedia.org/wiki/Rush_Hour_(puzzle)

FIGURE 7 : Jeux du taquin, de l'âne rouge (ou Klotski Puzzle) et Rush Hour.

En visitant moins de configurations, l'algorithme A∗ se révèle alors beaucoup plus efficace. Toutefois, il ne garantit pas toujours que la solution renvoyée soit la meilleure. Elle peut parfois n'être qu'une solution optimale au sens relatif du terme et non au sens absolu. Tout est question d'heuristique. Ajoutons que d'un point de vue algorithmique, la résolution de ces jeux est loin d'être simple! Ainsi, trouver la solution, c'est-à-dire un plus court chemin, à une configuration de taquin $n \times n$ est un problème NP-difficile. Seule la vérification d'une solution entre dans la classe P[11].

Parmi les applications, on peut également citer les déplacements sur grille, comme dans les jeux vidéos. La Figure 8 illustre la mise en oeuvre des deux algorithmes pour déterminer un plus court chemin entre un point de départ, situé en bas à gauche de chaque grille, et un point d'arrivée, situé en haut à droite de chaque grille. Citons enfin le domaine de l'Intelligence Artificielle qui fait un très large usage des algorithmes d'exploration.

11. Schématiquement, un problème algorithmique entre dans la *classe* P s'il existe un algorithme de complexité polynomiale qui le résout. Il entre dans la *classe* NP si on ne peut seulement que vérifier la complexité polynomiale d'une solution candidate. Un problème est dit NP-difficile si tout problème de la classe NP peut s'y ramener via une transformation appelée de *réduction polynomiale*. Si en outre, le problème lui-même est NP, on le qualifie alors de NP-complet. L'une des questions fondamentales actuelles de l'informatique est de savoir si P et NP sont une seule et même classe de complexité. Plus d'informations sont disponibles à ce sujet sur https://fr.wikipedia.org/wiki/Problème_NP-complet.

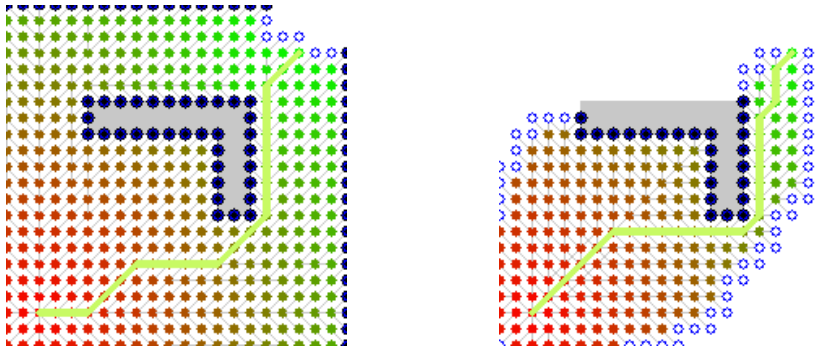


FIGURE 8 : Ensembles des sommets explorés lors du parcours d'une grille avec Dikjstra (à gauche) et A* (à droite).

SOLUTIONS DES EXERCICES