P2I: Initiation à la planification en Robotique

auteur : Jilles S. Dibangoye
enseignant : Aurélien Delage

Problématique Le but de ce TD/TP est de s'initier quelque peu aux arcanes de la planification en Robotique... L'idée est d'expérimenter les algorithmes « de parcours en largeur, de Dijkstra et A* ».

1 Prélude

Pour récupérer les sources python pour le TP, ouvrez un terminal et taper :

```
mkdir p2iPriseDecision
cd p2iPriseDecision
git clone  https://github.com/aurelienDelageInsaLyon/P2I6
cd P2I6
```

et installer le module networkx de visualisation de graph :

```
pip3 install networkx
```

Le langage de programmation est python, vous pouvez donc taper « python3 [...] » pour exécuter les fichiers .py.

2 Première partie

En robotique, nous recherchons souvent des chemins d'une origine vers une destination selon différents critères (*e.g.*, la distance ou le temps). La carte brute (Figure 1) illustre parfaitement les situations auxquelles nous sommes confrontés. À savoir, comment trouver un chemin en partant d'une position (*e.g.*, étoile) vers une autre (*e.g.*, croix).

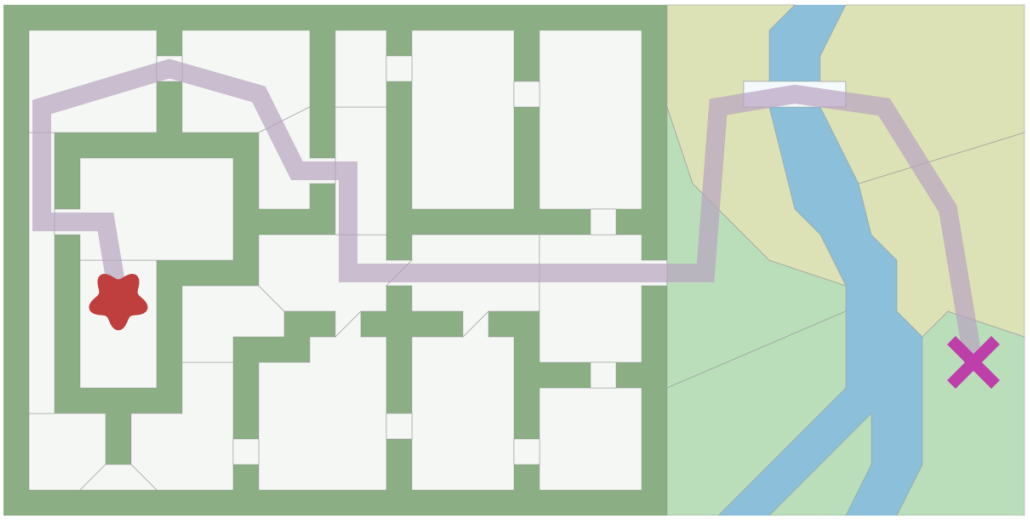

FIGURE 1 – Carte : l'étoile est l'origine et la croix est la destination.

Afin de trouver ce chemin, nous pouvons utiliser un algorithme de recherche arborescente ou « *graph search algorithm* ». L'algorithme A* est un choix populaire pour la recherche arborescente. Avant d'y arriver, nous verrons successivement l'algorithme de parcours en largeur, en profondeur, Dijkstra, et enfin A*.

2.1 Représentation graphique

La première chose à faire lorsqu'on cherche un chemin est de comprendre l'environnement et de le représenter de façon adéquate. Les algorithmes de recherche arborescente repose pour l'essentiel sur une représentation de l'environnement sous forme de graphe.

> **Définition 1.** Un graphe est donné par une paire $G \equiv (V, E)$, où V est l'ensemble des positions (« *noeuds* ») et E l'ensemble des connections (« *arêtes* ») du graphe.

Une illustration d'un graphe issue de la carte (Figure 1) est donnée en Figure 2, où les étiquettes des noeuds sont omis :

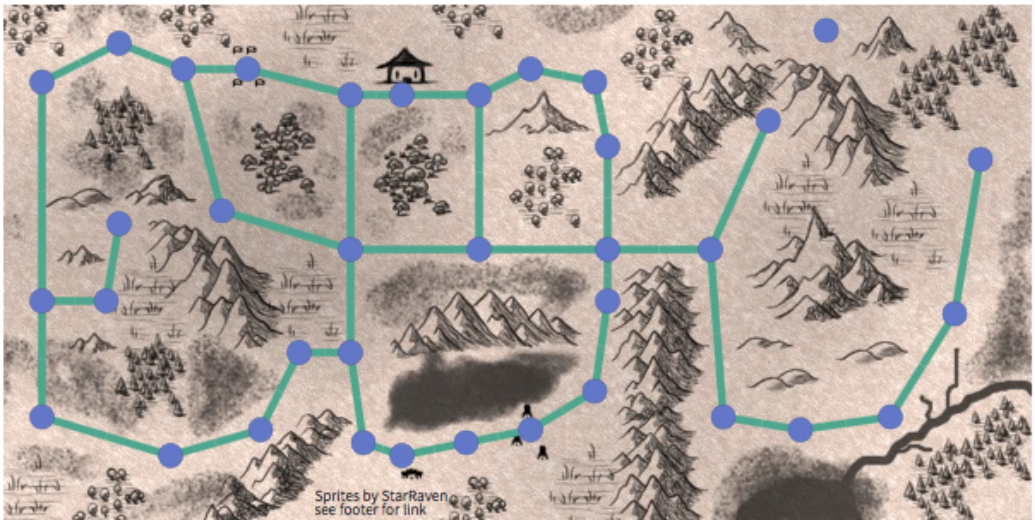

FIGURE 2 – Graphe dans le monde réel : les positions sont représentées par les cercles bleus.

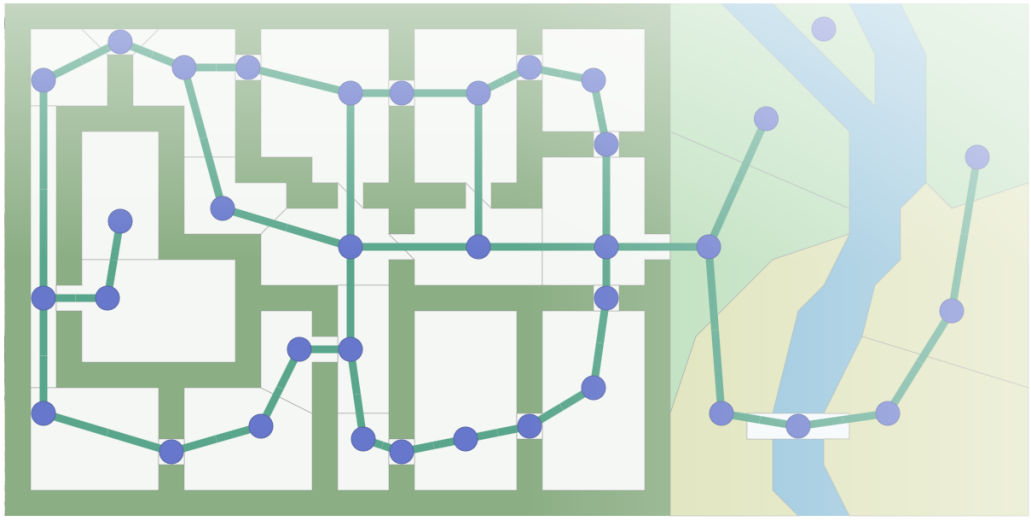

FIGURE 3 – Graphe dans un monde artificiel : les positions sont représentées par les cercles bleus.

Définition 2. Une grille est un graphe où tous les noeuds sont à équidistance des noeuds adjacents, *i.e.*, connectés par une arête.

Figure 4 propose une représentation sous forme de grille de la carte de

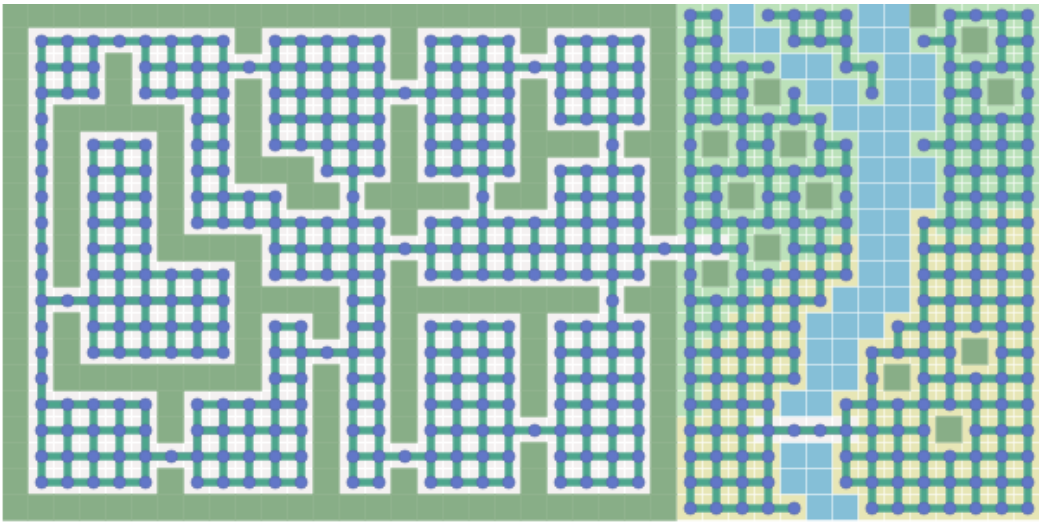

FIGURE 4 – Grille : les positions sont représentées par les cercles bleus.

navigation illustrée Figure 3.

> **Question 1.** Discuter des avantages et inconvénients des deux représentations.

2.2 Implémentation des représentations graphiques

Il est désormais temps de représenter sur ordinateur nos représentations graphiques sur papier. Les implémentations dépendent bien souvent de l'algorithme utilisé.

Un graphe est un objet mathématique composé de noeuds et arêtes. Ces noeuds sont connectés les uns aux autres via des arêtes. Ainsi pour concevoir un graphe, il nous faut :

— Un ensemble de noeuds du graphe
— Un ensemble d'arêtes issues de chaque noeud

> **Question 2.** Instancier et afficher le graphe illustré Figure 5 en utilisant les sources python fournies, en particulier le fichier simpleGraph.py. **Penser à bien regarder le constructeur de la classe SimpleGraph.**

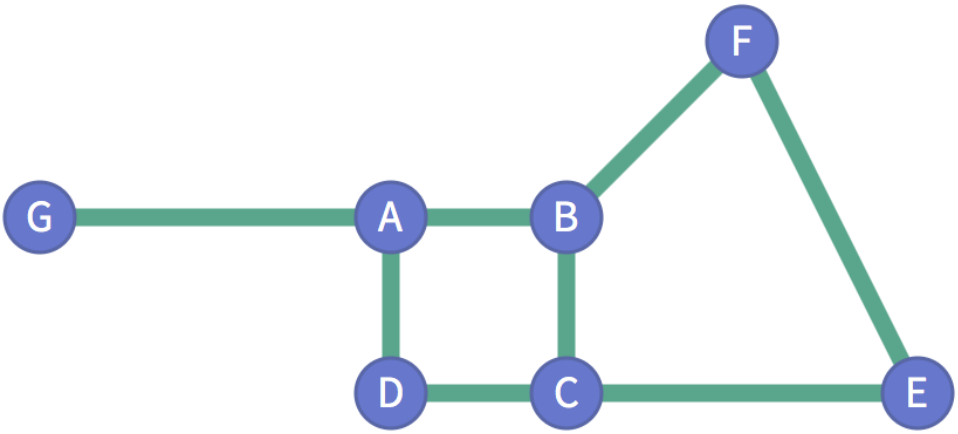

FIGURE 5 – Graphe théorique.

2.3 Conclusion

Nous avons étudié comment passer d'une carte à une représentation sur papier, puis vers une représentation numérique via une implémentation de graphes dans le langage de votre choix.

3 Seconde partie

Nous allons maintenant étudier les algorithmes de recherche arborescente : « parcours en largeur, Dijkstra et A* ». Par la suite, vous devrez modifier le fichier SquareGrid.py.

3.1 Quelques notes sur les structures Python

	heapq (pour Dijkstra & A*)	{}	[]
Sens	File de priorité	Dictionnaire	Tableau dynamique
Ajout	tab.heappush(…)	dict[key]=value	tab.append(…)
Acces	tab.heappop()	dict[Key]	tab[…]
Test de présence	—	in dict.Keys()	—

TABLE 1 – Structures de données utiles.

3.2 Algorithme de parcours en largeur

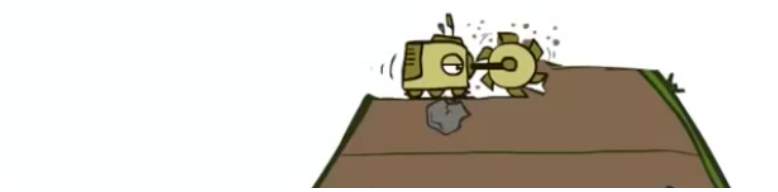

FIGURE 6 – Illustration de l'algorithme BFS.

L'algorithme de parcours en largeur est aussi appelé « *breadth-first-search* ». Nous utiliserons l'acronyme BFS par la suite.

L'algorithme BFS requiert :

— Un **graphe** : une structure de données capable de fournir les noeuds adjacents de tout noeud.
— Des **identifiants** (par exemple entier, caractère, tuple, etc.) qui étiquettent les noeuds du graphe.
— Une **file** : une structure de données utilisée par BFS pour ordonner le traitement des noeuds
— Un **algorithme** capable de prendre en entrée un graphe, un noeud source et d'un noeud destination, et calculer un chemin.

L'algorithme BFS part d'un noeud source s. Il liste **d'abord** les noeuds adjacents à s pour ensuite les explorer un par un. La file requise permet de sélectionner le prochain noeud à explorer. Le noeud en-tête de liste est le prochain noeud à explorer, ses noeuds adjacents seront placés en fin de liste. Les noeuds déjà visités sont marqués afin d'éviter qu'un même nœud soit exploré plusieurs fois. Dans le cas particulier d'un arbre, le marquage n'est pas nécessaire.

Pseudocode de BFS :

1. Mettre le noeud source dans la file.
2. Retirer le noeud du début de la file pour l'examiner.
3. Mettre tous les voisins (noeuds adjacents) non explorés en fin de file.
4. Si la file n'est pas vide reprendre à l'étape 2.

Question 3. Completer les sources fournies de l'algorithme BFS. Tester votre implémentation graphe à la méthode *test_bfs()* qui est fournie.

Question 4. Que peut-on dire de cet algorithme, est-il *terminal* [a], *complet* [b], *optimal* [c] ?

a. La **terminaison** garantie que l'algorithme terminera en temps fini.
b. La **complétude** garantie que, pour un espace donné, l'algorithme, s'il termine, donnera une solution pour chacune des entrées.
c. L'**optimalité** garantie que si l'algorithme termine en donnant une solution, alors cette solution est la meilleure selon un critère objectif prédéfini.

Question 5. Compléter une méthode de déterminisme de plus court chemin [a] sachant un noeud source et un noeud destination, en partant de l'algorithme de BFS. En d'autres termes, implémenter la méthode *recontruct_path*.

a. Une astuce consiste à stocker au cours de l'algorithme BFS, le meilleur prédécesseur de chaque noeud exploré.

3.3 Algorithme Dijkstra

Nous allons maintenant ajouter de la complexité à l'algorithme de recherche arborescente BFS. Nous allons traiter les noeuds dans un autre ordre meilleur que « *first-in, first-out* ». Ci-dessous les éléments dont nous aurons besoin :

1. Le *graphe* doit savoir le coût (*i.e.*, la durée, la distance) des déplacements ou des arêtes.
2. La *file* doit retourner des noeuds dans un ordre différent.
3. La *recherche arborescente* doit se souvenir des poids des arêtes issues du graphe et les rendre accessibles à la file.

Le reste de cette section détaille chacune des composantes (ci-dessus) nécessaires mais pas suffisantes pour l'implémentation de l'algorithme de Dijkstra. Cette section se termine par un descriptif de l'algorithme de Dijkstra.

3.3.1 Graphes pondérés

Nous devons modifier en profondeur notre structure de graphe. En particulier, il est possible (mais pas nécessaire) de modifier la représentation des noeuds afin d'y stocker leurs coordonnées. Dans tous les cas, il faut modifier le graphe afin d'y ajouter le poids associé à chaque arête. Un graphe pondéré est un graphe muni d'une méthode associant un poids à chaque arête du graphe.

> **Note.** Prenez connaissance de la méthode « `cost(from_node, to_node)` » capable de retourner le poids[a] de l'arête entre les noeuds `from_node` et `to_node`.
>
> *a*. Une implémentation possible est celle de la distance euclidienne entre deux paires de coordonnées associées à deux noeuds.

3.3.2 Files avec priorités

Nous avons également besoin de files avec priorités. Cette file sert à trier les noeuds stockés dans l'ordre croissant des coûts associés.

> **Note.** Python offre une telle « structure », via la library « heapq ». L'utilisation de cette librairie impose que des fonctions de comparaison sur les classes des objects stockés dans le tableau sont implémentées. C'est le cas de la classe « WeightedVertex » définie dans WeightedVertex.py.

Un exemple d'utilisation d'une file de priorité stockant des entiers :

```
1
2 import heapq;
3
4 tab_int = [ 1, 10, 5, 3, 4, 7, 6, 9, 8 ];
5
6 heapq.heapify(tab_int);
7 print(tab_int)#tab_int n'est pas trie dans un ordre naturel,
    mais trie au sens d'un arbre binaire utilise par heapq pour
    implementer sa file de priorite
8
```

```
#utiliser heapq.heappush(tab_int,item) afin d'inserer l'objet item
heapq.heappush(tab_int,2)
print(tab_int)

#utiliser heapq.heappop(tab_int,item) afin d'obtenir l'item le plus prioritaire
p=heapq.heappop(tab_int)
print(p)
print(tab_int)
```

Listing 1 – Utilisation de la librairie heapq

3.3.3 Principe de l'algorithme de Dijkstra

Le principe de l'algorithme de Dijkstra est de trouver le chemin ayant le poids « *cumulé* » le plus faible entre un noeud source et un noeud destination, sachant que le poids « *cumulé* » d'un chemin est la somme des poids des arêtes qui le composent.

Les étapes de l'algorithme de Dijkstra sont :

1. On affecte le poids 0 au sommet origine (disons s), et on attribue provisoirement le poids ∞ aux autres noeuds.
2. Répéter les opérations suivantes tant que le noeud destination d n'est pas affecté d'un poids définitif
   (a) Parmi les noeuds dont le poids n'est pas définitivement fixé choisir un noeud x de poids p minimal. Marquer définitivement ce noeud x affecté du poids $p(x)$.
   (b) Pour tous les noeuds y qui ne sont pas définitivement marqués, adjacents au dernier noeud fixé x :
   — Calculer la somme $p(y)$ du poids de x et du poids de l'arête reliant x à y — *i.e.*, $p(y) = \min\{p(y), cost(x, y) + p(x)\}$
   — Indiquer entre parenthèses le sommet x pour se souvenir de sa provenance.

3.3.4 Illustration sur un exemple

Considérons l'exemple Figure 7, avec comme noeud source G et noeud destination E. Pour faciliter la recherche de plus court chemin, il est commode d'illustrer l'algorithme de Dijkstra via un tableau à deux dimen-

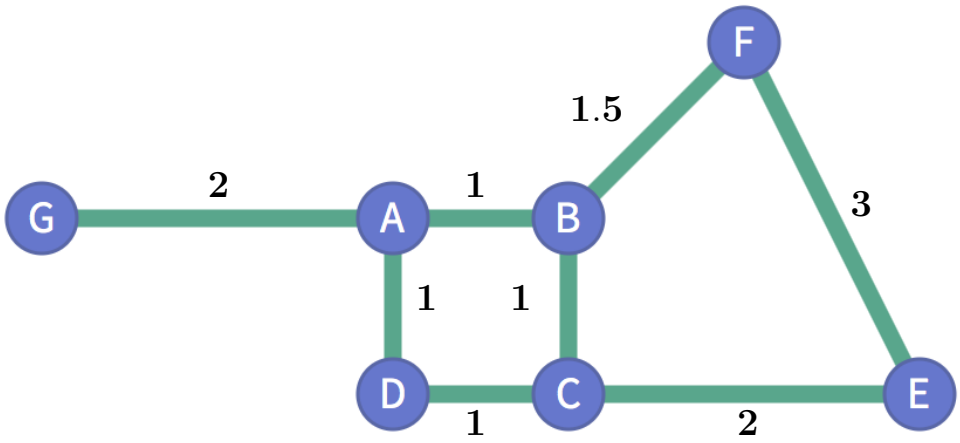

FIGURE 7 – Graphe pondéré utilisé pour illustrer l'algorithme Dijkstra.

sions : (1) l'ensemble des noeuds correspond aux colonnes ; et (2) l'estimation à chaque étape du poids cumulé du noeud source à chacun des noeuds du graphe, correspond à une ligne. Noter que cette estimation est une valeur pessimiste du poids minimum réel.

1. On affecte le poids 0 au sommet origine G, et on attribue provisoirement le poids ∞ aux autres noeuds.

	A	B	C	D	E	F	G	Noeuds sélectionnés
étape initiale	∞	∞	∞	∞	∞	∞	0	G est de poids 0.

TABLE 2 – Algorithme de Dijkstra en marche !

2. Parmi les noeuds $\{G\}$ dont le poids n'est pas définitivement fixé choisir un noeud x de poids minimal, i.e., $p = \min_{x \in \{G\}} p(x)$. Marquer définitivement ce noeud G affecté du poids $p(G) = 0$.

	A	B	C	D	E	F	G	Noeuds sélectionnés
étape initiale	∞	∞	∞	∞	∞	∞	0	G est de poids 0.
$p(G) = 0$ définitif	∞	∞	∞	∞	∞	∞	–	A est de poids 2.

TABLE 3 – Algorithme de Dijkstra en marche !

3. Pour tous les noeuds $y \in \{A\}$ qui ne sont pas définitivement marqués, adjacents au dernier noeud fixé G :

— Calculer la somme $p(A)$ la somme du poids de G et du poids de l'arête reliant G à A — *i.e.*, $p(A) = \min\{p(A), cost(G, A)+p(G)\} = 2$
— Indiquer entre parenthèses le sommet G pour se souvenir de sa provenance.

	A	B	C	D	E	F	G	Noeuds sélectionnés
étape initiale	∞	∞	∞	∞	∞	∞	0	G est de poids 0.
$p(G) = 0$ définitif	2	∞	∞	∞	∞	∞	–	A est de poids 2.

TABLE 4 – Algorithme de Dijkstra en marche !

4. Parmi les noeuds $\{A\}$ dont le poids n'est pas définitivement fixé choisir un noeud x de poids minimal, i.e., $p = \min_{x\in\{A\}} p(x)$. Marquer définitivement ce noeud A affecté du poids $p(A) = 2$.
5. Pour tous les noeuds $y \in \{B, D\}$ qui ne sont pas définitivement marqués, adjacents au dernier noeud fixé A :
   — Calculer la somme $p(x)$ la somme du poids de $x \in \{B, D\}$ et du poids de l'arête reliant A à $x \in \{B, D\}$ — *i.e.*, $p(B) = \min\{p(B), cost(A, B)+p(A)\} = 3$ et $p(D) = \min\{p(D), cost(A, D) + p(A)\} = 3$
   — Indiquer entre parenthèses le sommet A pour se souvenir de sa provenance.
6. Repéter cette procédure tant que le noeud destination E n'est pas affecté d'un poids définitif.

Suivant l'origine de chaque noeud marqué définitivement, il est possible de reconstruire le chemin de poids minimal. Soit G-A-B-C-E de poids cumulé 6.

3.3.5 Questions sur l'algorithme de Dijkstra

Question 7. Compléter l'algorithme Dijkstra. Vous pouvez utiliser, ou non, une liste de priorité. Tester votre implémentation grâce à la méthode *test_dijsktra()* qui vous est fournie.

Question 8. Que peut-on dire de cet algorithme, est-il *terminal* [a], *complet* [b], *optimal* [c] ?

a. La **terminaison** garantie que l'algorithme terminera en temps fini.
b. La **complétude** garantie que, pour un espace donné, l'algorithme, s'il termine, donnera une solution pour chacune des entrées.
c. L'**optimalité** garantie que si l'algorithme termine en donnant une so-

	A	B	C	D	E	F	G	Noeuds sélectionnés
étape initiale	∞	∞	∞	∞	∞	∞	0	G est de poids 0.
$p(G) = 0$ définitif	2	∞	∞	∞	∞	∞	–	A est de poids 2.
$p(A) = 2_G$ définitif	–	∞	∞	∞	∞	∞	–	B et D sont de poids 3.

TABLE 5 – Algorithme de Dijkstra en marche !

	A	B	C	D	E	F	G	Noeuds sélectionnés
étape initiale	∞	∞	∞	∞	∞	∞	0	G est de poids 0.
$p(G) = 0$ définitif	2	∞	∞	∞	∞	∞	–	A est de poids 2.
$p(A) = 2_G$ définitif	–	3	∞	3	∞	∞	–	B et D sont de poids 3.

TABLE 6 – Algorithme de Dijkstra en marche !

	A	B	C	D	E	F	G	Noeuds sélectionnés
étape initiale	∞	∞	∞	∞	∞	∞	0	G est de poids 0.
$p(G) = 0$ définitif	2	∞	∞	∞	∞	∞	–	A est de poids 2.
$p(A) = 2_G$ définitif	–	3	∞	3	∞	∞	–	B et D sont de poids 3.
$p(B) = 3_A$ définitif	–	–	4	3	∞	4.5	–	C et F sont de poids 4 et 4.5.
$p(D) = 3_A$ définitif	–	–	4	–	∞	4.5	–	C reste de poids 4.
$p(C) = 4_B$ définitif	–	–	–	–	6	4.5	–	E est de poids 6.
$p(F) = 4.5_B$ définitif	–	–	–	–	6	–	–	E reste de poids 6.
$p(E) = 6_C$ définitif	–	–	–	–	–	–	–	Fin de l'algorithme.

TABLE 7 – Algorithme de Dijkstra en marche !

lution, alors cette solution est la meilleure selon un critère objectif prédéfini.

3.4 Algorithme A*

L'algorithme A* est presqu'identique à Dijkstra, sauf qu'il faut y ajouter une heuristique. Une heuristique est une fonction connue qui associe à tout noeud une *estimation optimiste*[1] du poids cumulé de ce noeud y au noeud destination d — noté $h(y)$. A* maintient l'heuristique, noté $f(y)$, en tout noeud y comme suit. Pour tous noeuds x (marqué définitivement) et y (non marqué et adjacent à x) :

$$f(y) = p(y) + cost(x, y) + h(y).$$

1. Une estimation optimiste dans un problème de minimisation de poids est une estimation plus petite que le poids cumulé rée

Question 9. Compléter une méthode « `heuristique(from_node)` » capable de retourner une estimation optimiste du poids cumulé de ce noeud au noeud destination [a].

a. Une implémentation possible est celle des distances entre les deux noeuds — *i.e.*, distance euclidienne, de Manhattan, etc.

3.4.1 Le principe de A*

Les étapes de l'algorithme de A* sont :

1. On affecte le poids cumulé $p(s) = 0$ (de l'origine au noeud courant) au sommet origine (disons s), et on attribue provisoirement le poids $p(x) = \infty$ aux autres noeuds x.
2. Mettre le noeud source dans la file de priorité, où les noeuds sont ordonnés suivant leurs poids cumulé $f(x)$ (du noeud courant à la destination).
3. Si la file de priorité est vide alors il n'existe pas de chemin de la source à la destination ; l'algorithme est terminé.
4. Sinon
   (a) choisir un noeud x de poids cumulé $f(x)$ minimal (en cas d'égalité sélectionner arbitrairement)
   (b) s'il s'agit de la destination, l'algorithme a trouvé un plus court chemin ; et termine.
   (c) Sinon
      i. si le noeud n'est pas marqué : marqué le et ajouter les noeuds adjacents dans la file.
      ii. sinon, revenir à l'étape 2.

L'ajout des noeuds adjacents dans le file de priorité suit les étapes suivantes :

1. Pour tout les noeuds adjacents y au noeud courant x
   (a) Si y n'est pas marqué
      i. Créer le noeud en y attachant son parent,
      ii. Actualiser ses poids cumulés — *i.e.*,

$$p(y) = p(x) + cost(x, y)$$

et

$$f(y) = p(y) + h(y).$$

(b) Ajouter y dans la file de priorité suivant le poids cumulé $f(y)$.

Considérons l'exemple Figure 7, avec comme noeud source G et noeud destination E. Pour faciliter la recherche de plus court chemin, il est commode d'illustrer l'algorithme A* via un tableau à deux dimensions : (1) l'ensemble des noeuds correspond aux colonnes ; et (2) l'estimation des poids cumulés $p(x)/f(x)$ à chaque étape, correspond à une ligne.

A	B	C	D	E	F	G
3.5	3	2	3	0	3	4.5

TABLE 8 – Heuristique optimiste $h(x)$.

Question 10. Compléter l'exécution de l'algorithme A* via le Tableau 9 ci-dessous.

	A	B	C	D	E	F	G	Noeuds sélectionnés
étape initiale	∞/∞	∞/∞	∞/∞	∞/∞	∞/∞	∞/∞	$0/0$	G est de poids $0/0$.

TABLE 9 – Algorithme A* en marche !

3.4.2 Questions sur l'algorithme A*

Question 11. Compléter l'algorithme A*.

Question 12. Que peut-on dire de cet algorithme, est-il *terminal* [a], *complet* [b], *optimal* [c] ?

a. La **terminaison** garantie que l'algorithme terminera en temps fini.
b. La **complétude** garantie que, pour un espace donné, l'algorithme, s'il termine, donnera une solution pour chacune des entrées.
c. L'**optimalité** garantie que si l'algorithme termine en donnant une solution, alors cette solution est la meilleure selon un critère objectif prédéfini.

4 Conclusion

Question 14. Comparer les propriétés des trois algorithmes étudiés, selon leurs performances sur l'exemple de graphe utilisé jusqu'ici. Sous quelle(s) condition(s) A* et Dijkstra sont-ils équivalents ? Quel est le meilleur algorithme ?