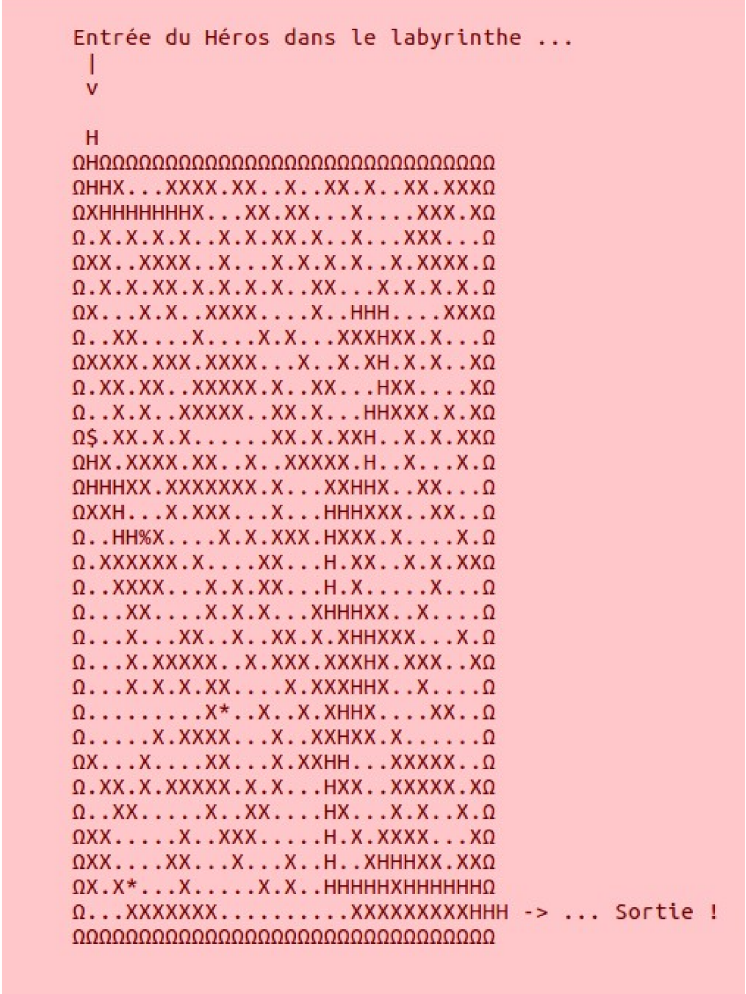
Rapport du Projet IPI : chemins de poids optimum.

Adrien BLASSIAU

13 janvier 2018



Introduction

L'objectif de ce projet d'algorithmie est d'élaborer plusieurs programmes, de plus en plus sophistiqués, de recherche de chemin de poids minimum dans des graphes orientés ou non entre un nœud de départ s et un nœud d'arrivée t.

Dans un premier temps, on implante l'algorithme de parcours en largeur qui trouve un chemin de poids minimum où tous les arcs sont de même poids 1 (**Exercice 1**).
Ensuite, on réalise l'algorithme de Dijkstra qui est une généralisation de l'algorithme précédent car les arcs peuvent être évalués avec des poids positifs cette fois (**Exercice 2**). Mais l'algorithme de Dijkstra est en fait un cas particuliers de l'algorithme A* où l'heuristique est nulle.
Ainsi dans un troisième temps, on implémente ce dernier qui, en choisissant un bonne fonction heuristique, permet de trouver le chemin optimal dans un labyrinthe dans la plupart des cas, tout en étant bien plus rapide que les deux autres algorithmes.(**Exercice 3**).

1 Exercice 1 : Algorithme de parcours en largeur

On considère un graphe orienté G = (V,A), dans lequel on cherche un chemin avec un nombre minimum d'arcs entre le nœud d'entrée s et le nœud de sortie t, tous les arcs sont donc de poids 1. L'ensemble des fonctions et structures se trouveront dans le fichier parcourslargeur.c et leur prototype dans le fichier parcourslargeur.h.

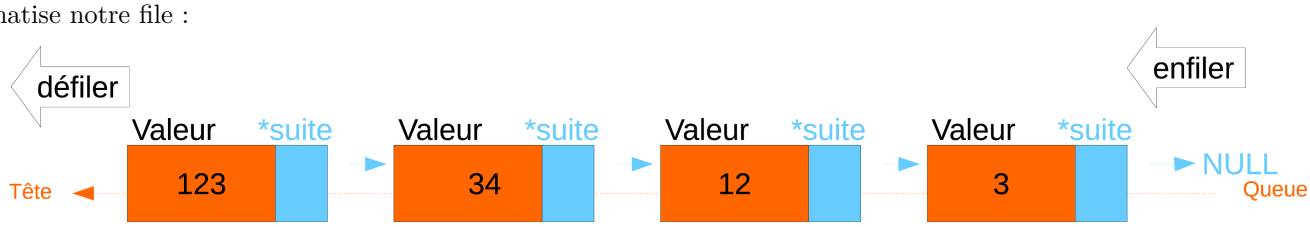
1.1 Démarche

Le principe de l'algorithme de parcours en largeur est simple. On commence par énumérer les successeurs S_1 du sommet de départ. Ensuite, on énumère les successeurs S_2 des nœuds de S_1 pas encore énumérés et ainsi de suite tant que l'on n'énumère pas le sommet d'arrivée ou que l'on n'a pas énuméré tout les sommets possibles. À chaque fois que l'on énumère un nœud, il est marqué et on retient son prédécesseur.
Si l'on énumère le nœud d'arrivée, le programme s'arrête et on remonte au nœud de départ en regardant de proche en proche les prédécesseurs de chaque nœud, en commençant par le prédécesseur du nœud d'arrivée. On obtient ainsi le chemin optimal mais de l'arrivée au départ, il suffit alors d'inverser ce chemin pour obtenir le chemin optimal recherché du départ à l'arrivée.
Si plus aucun sommet n'est à énumérer, le sommet d'arrivée n'est pas connecté au sommet de départ et appartient à une autre composante connexe de G. Il n'existe pas de sommet de s à t.

1.2 Présentation des structures de données principales utilisées

1.2.1 La file

La file, appelée queue en anglais, repose sur le principe du "premier entré, premier sorti" aussi appelé FIFO (Firs-In First-Out). Cela signifie que le premier élément rentré dans la file sera le premier à en sortir. Voici comment se schématise notre file :



En C, notre file est modélisée par un ensemble de maillons pris entre un maillon tête et un maillon queue. Les deux opérations que l'on effectue sur notre file sont : ajouter un élément (enfiler) et retirer un élément (défiler).
Lorsqu'on enfile un maillon, ce nouveau maillon pointe sur l'ancienne queue de la file est devient ainsi la nouvelle queue. Lorsqu'on défile un maillon, qui était la tête, le maillon suivant devient la tête. La présence d'un maillon tête et d'un maillon queue permet d'accéder facilement au premier et au dernier élément de la file et donc d'enfiler et de défiler facilement les éléments de la file.

1.2.2 Le tableau des prédécesseurs

Pour sauvegarder les prédécesseurs de chaque sommet rencontré et ainsi pouvoir reconstruire le chemin si il existe, on utilise un simple tableau de taille n avec n étant le nombre de sommets du graphe. Chaque case d'indice i du tableau représente le nœud d'indice i+1, son contenu est le numéro de son prédécesseur. Au départ le tableau est rempli de zéros.

1.3 Pseudo-code du programme

Voici le pseudo-code de la fonction principale du programme :

Entrées : Un graphe orienté G = (V,A) et deux nœuds s et t.
Sorties : Un tableau valant NULL si aucun chemin trouvé, sinon avec les différents nœuds du chemin trouvé rangés du départ à l'arrivée.

01 : enfiler s dans la file
02 : marquer que le prédécesseur de s est -1 dans le tab_pred, c'est à dire que s n'a pas de prédécesseur
03 : **Tant que** la tête ne vaut pas NULL ou que la queue ne vaut pas NULL ou que t n'a pas de prédécesseur dans tab_pred **Faire**
04 : défiler l'élément u en tête de file
05 : **Pour** chaque successeur v de u pas encore marqué dans tab_pred **Faire**
06 : enfiler v
07 : marquer que son prédécesseur est u dans tab_pred
08 :
09 : **Si** la tête et la queue valent NULL **Alors**
10 : écrire "Not connected"
11 : tab_fin vaut NULL
12 : **Sinon**
13 : récupérer le chemin de s à t dans un tableau tab_fin en remontant tab_pred
14 :
15 : **Renvoyer** tab_fin qui contient le chemin de s à t

1.4 Mise en œuvre sur un exemple

On choisit d'expliquer le fonctionnement du programme sur le graphe suivant, qui correspond à la matrice d'adjacence de l'exemple 1 :

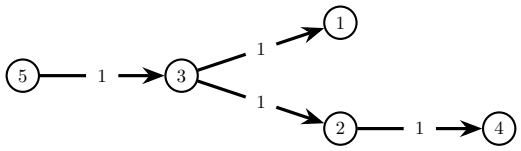


Table 1 – Déroulé de l'algorithme

Action	État de la file	État du tableau des prédécesseurs
On enfile 5	5 -> NULL	[0,0,0,0,-1]
On défile 5	NULL	[0,0,0,0,-1]
On enfile 3	3 -> NULL	[0,0,5,0,-1]
On défile 3	NULL	[0,0,5,0,-1]
On enfile 1	1 -> NULL	[3,0,5,0,-1]
On enfile 2	1 -> 2 -> NULL	[3,3,5,0,-1]
On défile 1	2 -> NULL	[3,3,5,0,-1]
On défile 2	NULL	[3,3,5,0,-1]
On enfile 4	4 -> NULL	[3,3,5,2,-1]

On obtient comme tableau des prédécesseurs [3,3,5,2,-1]. On s'arrête car 4 possède un prédécesseur. On remonte maintenant le tableau des prédécesseurs : 4 a pour prédécesseur 2 qui a pour prédécesseur 3 qui a pour prédécesseur 5 qui n'a pas de prédécesseur. Le chemin est donc 5-3-2-4.

1.5 Bilan

J'ai choisi la file car il était nécessaire de hiérarchiser dans leur ordre d'arrivée les différents successeurs visités de chaque sommet. De plus, à chaque fois qu'un nœud est défilé, tout ses successeurs sont enfilés. Le déplacement de l'information est donc important, il semble donc primordial que notre structure puisse évoluer rapidement de manière dynamique. Ainsi, après avoir implanté une file sans pointeur sur la tête de file, je me suis rendu compte qu'il était mieux de retenir la position de cette tête car pour y accéder, il fallait traverser toute la file ce qui était assez coûteux.

Ainsi l'introduction d'un pointeur sur la tête de file en plus du pointeur déjà existant sur la queue de file a permis l'accès au premier et au dernier élément rapidement, en un 0(1).
Tous les tests passent en un temps moyen total de 0,3 secondes chez moi.

2 Exercice 2 : Algorithme de Dijkstra

On considère un graphe orienté G = (V,A), dans lequel on cherche un chemin de poids minimum entre l'entrée s et la sortie t, les arcs étant évalués avec des poids positifs. Cette algorithme ressemble fortement au précédent sauf que cette fois ci, le poids des arcs est pris en compte. L'ensemble des fonctions et structures se trouvent dans le fichier dijkstra.c et leur prototype dans le fichier dijkstra.h.

2.1 Démarche

Là encore le principe de l'algorithme de Dijkstra est simple. On commence par énumérer les successeurs S_1 du sommet de départ que l'on marque. Parmi ces successeurs, on choisit celui qui est le plus proche en coût du sommet de départ s et on le marque. Pour l'algorithme précédent, seule la distance était prise en compte, c'est à dire le nombre d'arcs. Ici, la somme du poids des arcs, c'est à dire le coût, est pris en compte. Ensuite, on énumère les successeurs S_2 pas encore marqués du nœud choisi dans S_1 et on sélectionne le nœud de coût le plus faible parmi tous les successeurs énumérés depuis le début, sans prendre en compte ceux marqués. Le coût peut être mis à jour si on rencontre une nouvelle fois un successeur déjà énuméré avec un coût plus faible. On réitère jusqu'à ce que l'on sélectionne le nœud de sortie t ou qu'il n'y ait plus aucun nœud à énumérer.

2.2 Présentation de la structure de données utilisée

2.2.1 Le tableau de structure de nœuds

Cette fois ci, on construit une structure de nœuds qui contient le coût du nœud au départ (coût), le nœud prédécesseur qui minimise cette distance (prédécesseur) et une indication si ce nœud a déjà été marqué ou non (marque). Ensuite, on fabrique un tableau où chaque case d'indice i représente le nœud de numéro i+1 décrit par une structure de type nœud.

2.3 Pseudo-code du programme

J'ai suivi le cheminement conseillé par le sujet, le pseudo-code est donc le même.

2.4 Mise en œuvre sur un exemple

On choisit d'expliquer le fonctionnement du programme sur le graphe suivant, qui correspond à la matrice d'adjacence de l'exemple 1 :

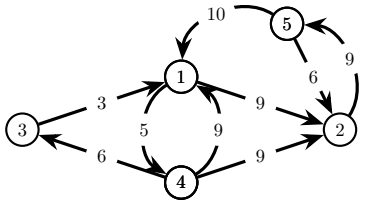


Table 2 – Déroulé de l'algorithme

Action	État des coûts	État des prédécesseurs	État des marqués
On choisit 3	[∞,∞,0,∞,∞]	[51,51,0,51,51]	[0,0,1,0,0]
On trouve 1	[3,∞,0,∞,∞]	[3,51,0,51,51]	[0,0,1,0,0]
On choisit 1	[3,∞,0,∞,∞]	[3,51,0,51,51]	[1,0,1,0,0]
On trouve 4	[3,∞,0,8,∞]	[3,51,0,1,51]	[1,0,1,0,0]
On trouve 2	[3,12,0,8,∞]	[3,1,0,1,51]	[1,0,1,0,0]
On choisit 4	[3,12,0,8,∞]	[3,1,0,1,51]	[1,0,1,1,0]
On trouve 2	[3,12,0,8,∞]	[3,1,0,1,51]	[1,0,1,1,0]
On choisit 2	[3,12,0,8,∞]	[3,1,0,1,51	[1,1,1,1,0]

On obtient comme tableau des prédécesseur [3,1,0,1,∞]. On s'arrête car on a marqué 2. On remonte maintenant le tableau des prédécesseur : 2 à pour prédécesseur 1 qui à pour prédécesseur 3. Le chemin est donc 3-1-2.

2.5 Bilan

Dans la mesure où le pseudo-code était donné, cette exercice n'était pas le plus compliqué. Cependant, le choix de la structure de données était primordiale car de nombreuses informations étaient perpétuellement modifiées et il fallait une structure aux éléments faciles d'accès et qui puisse être aisément modifiée. J'ai choisi cette structure de tableau de nœuds car elle permettait de regrouper de manière synthétique toutes les informations nécessaires. Dans l'exercice 3, on verra que l'on aurait pu aussi utiliser une liste de priorité mais ce n'était pas nécessaire ici car le nombre de nœuds traités était faible et la recherche du min rapide. J'ai réservé cette structure à l'algorithme A *.
Tous les tests passent en un temps moyen total de 0,27 secondes chez moi.

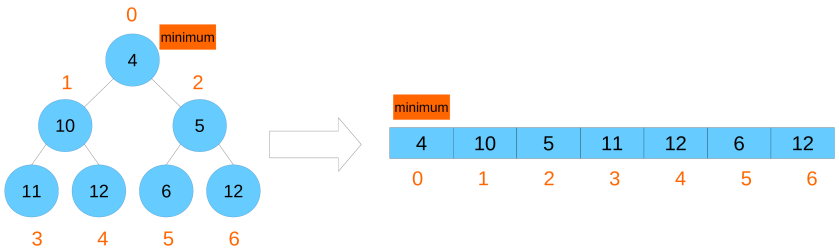
3 Exercice 3 : Résolution d'un labyrinthe avec Algorithme A *

On considère cette fois la carte d'un labyrinthe avec une entrée E positionnée en haut à gauche et une sortie S positionnée en bas à droite. On cherche un chemin pour se rendre de l'entrée E à la sortie S avec un temps imparti donné. L'algorithme utilisé ici est l'algorithme A *. L'ensemble des fonctions et structures se trouvent dans le fichier parcourslargeur.c, labyrinthe.c et aetoile.c et leur prototype dans le fichier parcourslargeur.h, labyrinthe.h et aetoile.h.

3.1 Présentation des structures de données principales utilisées

3.1.1 La file de priorité

On utilise une structure efficace qui met en avant le minimum des éléments qu'elle contient : **la file de priorité** (**priority queue** en anglais) implémentée en **tas** (**heap** en anglais) sous la forme d'un **tableau**. Définissons chacun de ces termes. **La file de priorité** est une file dans laquelle l'élément que l'on extrait est le plus petit ou le plus grand. Dans notre cas, ce sera le plus petit. La file est donc triée en amont, à chaque ajout d'un élément, d'où le terme de priorité. Ce type de file est implanté en **tas** : c'est une structure de données que l'on représente schématiquement sous forme d'arbre binaire. Dans cet arbre, la valeur contenue dans chaque nœud père est inférieure ou égale à la valeur de ses fils gauche et droit. On numérote les nœuds de haut en bas et de gauche à droite. Le tas est ainsi représenté réellement sous forme de **tableau** où le numéro des nœuds correspond aux indices du tableau. Voici un schéma récapitulatif :



Quelques petites formules sont utiles pour l'exploitation du tas :

Table 3 – Formules utiles du tas

Nœud	Indice
Racine (minimum)	0
Nœud	i
Parent	$\lfloor \frac{i-1}{2} \rfloor$
Fils gauche	$2i+1$
Fils droit	$2i+2$

Plusieurs opérations utiles dont nous aurons besoin peuvent être effectuées sur le tas :
— l'insertion d'un élément : on positionne le nouveau nœud dans la première position libre puis on le permute avec son père jusqu'à ce qu'il soit plus grand que son père. L'opération est en O(log n).
— l'extraction du maximum : on donne à la racine la valeur du dernier nœud, on supprime le dernier nœud puis on permute la valeur de la racine avec le plus petit de ses deux fils jusqu'à ce que ses deux fils soient plus grands. L'opération est en O(log n).
— la modification d'un élément : on modifie la valeur du nœud si cette valeur est plus petite que la précédente puis on le permute avec son père jusqu'à ce qu'il soit plus grand que son père. L'opération est en O(log n).

Notre tas est donc un tableau de structures info qui contient différentes informations sur chaque nœud du labyrinthe, dont le coût heuristique (cout_heuri), sera la valeur selon laquelle les nœuds seront triés. Ce tableau est lui même contenu dans une structure file_p qui contient aussi l'élément taille qui corespond au nombre d'éléments

contenus dans le tas.

3.1.2 La matrice de structure de nœuds de labyrinthe

On traduit le labyrinthe sous la forme d'une matrice de structure nœud_labyrinthe. Chaque structure correspond à une case du labyrinthe. Une structure se compose du type de la case, du numéro du nœud qui représente la case et d'une indication pour savoir si c'est le première fois que l'on rencontre cette case ou non (utile pour les téléporteurs).

3.1.3 La matrice d'adjacence

La matrice d'adjacence utilisée est un peu particulière. En effet, on remarque que certains labyrinthes possèdent plusieurs milliers de nœuds voir 1 000 000, ce qui nécessite l'utilisation d'une matrice d'adjacence de 1 000 000 * 1 000 000, chose que la machine ne peut réaliser. Or, une fois les cases du labyrinthe converties en nœud, chaque nœud peut avoir au maximum 8 voisins (en prenant en compte les cases de type téléporteur). Ainsi notre matrice d'adjacence n'est qu'une matrice qui recense les voisins de chaque nœud. De plus, afin de conserver un maximum d'information, on rajoute les positions x et y de nos nœuds dans le labyrinthe. Finalement, notre pseudo matrice d'adjacence a une taille de nombre de nœuds * 12 (car deux positions si téléporteurs), ce qui est envisageable de résoudre.

3.1.4 Le tableau des prédécesseurs

Pour sauvegarder les prédécesseurs de chaque sommet rencontré et ainsi pouvoir reconstruire le chemin s'il existe, on utilise un simple tableau de taille n, avec n le nombre de sommets du graphe. Chaque case d'indice i du tableau représente le nœud d'indice i+1, son contenu est le numéro de son prédécesseur. Au départ le tableau est rempli de zéros.

3.2 Démarche

Tout d'abord, on convertit le labyrinthe fourni en entrée sous la forme d'une matrice de char en un tableau de structure nœud_labyrinthe en donnant le même numero_nœud aux téléporteurs similaires. Ensuite, on réalise la matrice d'adjacence du labyrinthe en ne prenant que les nœuds qui ne sont pas des murs. Enfin, on applique l'algorithme A * sur la matrice d'adjacence dont voici le fonctionnement, fortement similaire à l'algorithme de Dijkstra.

On commence par énumérer les successeurs S_1 du sommet de départ que l'on marque. Parmi ses successeurs, on choisit celui qui possède le coût heuristique le plus faible et on le marque. *Mais qu'est ce que le coût heuristique ?*
Et bien c'est la somme du coût, le même que celui utilisé dans l'algorithme de Dijkstra et d'une heuristique qui estime le coût du chemin du nœud en question à l'arrivée. On utilise la distance de Manhattan qui est la distance minimale entre deux points A et B placés sur les nœuds d'un quadrillage en utilisant seulement des déplacements verticaux et horizontaux. Elle est donnée par la formule $dist(A, B) =\mid X_A - X_B \mid + \mid Y_A - Y_B \mid$.

Ensuite, on énumère les successeurs S_2 pas encore marqués du nœud choisi dans S_1 et on sélectionne le nœud de coût heuristique le plus faible parmi tous les successeurs énumérés depuis le début, sans prendre en compte ceux marqués. Le coût heuristique peut être mis à jour si on rencontre une nouvelle fois un successeur déjà énuméré avec un coût plus faible. On réitère jusqu'à ce que l'on sélectionne le nœud de sortie t ou qu'il n'y ait plus aucun nœud à énumérer.

Plus précisément, l'algorithme utilise l'algorithme A* dans un premier temps et si aucun chemin satisfaisant n'a été trouvé, il utilise l'algorithme A* avec une heuristique nulle, qui correspond en fait à une implémentation de l'algorithme Dijkstra ; mais comme les poids valent tous 1, on réalise en fait un parcours en largeur. De plus, on distingue deux cas, celui d'un labyrinthe avec porte et celui d'un labyrinthe sans porte. Si le labyrinthe est avec porte, on cherche tout d'abord un chemin qui passe par la porte. Pour cela, on cherche un chemin du début à la clé, de la clé à la porte puis de la porte à l'arrivée. Si ce chemin n'est pas satisfaisant, on cherche un chemin ne passant pas par la porte.

Enfin, une fois que l'on a obtenu le chemin, il faut le convertir en direction, ce qu'une fonction se charge de faire avec la matrice d'adjacence comme donnée d'entrée notamment.

3.3 Pseudo-code du programme

Voici le pseudo-code de l'algorithme A étoile :

Entrées : Une variable choix qui vaut 1 si l'heuristique est nulle, un graphe G = (V,A) représenté par sa matrice d'adjacence m, la taille du graphe, l'entrée s et la sortie t du labyrinthe, la taille n du labyrinthe et un pointeur sur le temps du parcours *temps_parcours.
Sorties : Un tableau valant NULL si aucun chemin trouvé, sinon avec les différents nœuds du chemin trouvé rangés

du départ à l'arrivée.

01 : insérer s dans le tas
02 : marquer que le prédécesseur de s est -1 dans le tab_pred, c'est à dire que s n'a pas de prédécesseurs
03 : **Tant que** la taille du tas n'est pas nulle ou que t n'a pas de prédécesseur dans tab_pred **Faire**
04 : extraire l'élément u du tas
05 : marquer u comme visité
06 : **Pour** chaque successeur v de u pas encore marqué dans tab_visite**Faire**
07 : calculer le coût heuristique de v
08 : **Si** v n'est pas encore présent dans le tas ou alors présent mais avec un coût supérieur **Alors**
08 : marquer que son prédécesseur est u dans tab_pred
09 : insérer v dans le tas ou modifier le tas si v déjà présent
10 :
11 : **Si** le tas à une taille inférieure à 0 **Alors**
12 : tab_fin vaut NULL
13 : **Sinon**
14 : récupérer le chemin de s à t dans un tableau tab_fin en remontant tab_pred
15 :
16 : **Renvoyer** tab_fin qui contient le chemin de s à t

3.4 Mise en œuvre sur un exemple

On choisit d'expliquer le fonctionnement du programme sur un graphe qui correspond au labyrinthe 1 :

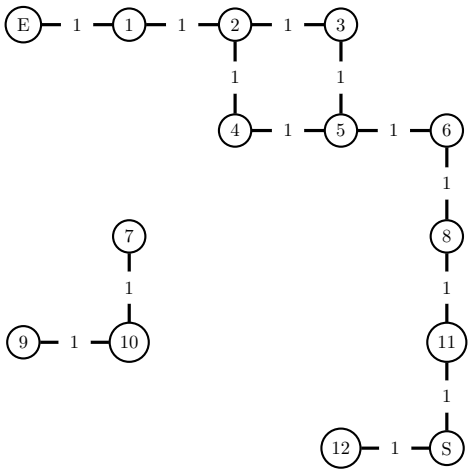


Table 4 – Déroulé de l'algorithme

Action	État du tas (coût heuristique/coût/heuristique)	État du tableau des prédécesseur
On insère E	[E (8/0/8)]	[-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
On extrait E	[]	[-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
On insère 1	[1 (8/1/7)]	[-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
On extrait 1	[]	[-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
On insère 2	[2 (8/2/6)]	[-1,0,1,0,0,0,0,0,0,0,0,0,0,0,0]
On extrait 2	[]	[-1,0,1,0,0,0,0,0,0,0,0,0,0,0,0]
On insère 3	[3 (8/3/5)]	[-1,0,1,2,0,0,0,0,0,0,0,0,0,0,0]
On insère 4	[4 (8/3/5),3 (8/3/5)]	[-1,0,1,2,2,0,0,0,0,0,0,0,0,0,0]
On extrait 4	[3 (8/3/5)]	[-1,0,1,2,2,0,0,0,0,0,0,0,0,0,0]
On insère 6	[6 (8/5/3),3 (8/3/5)]	[-1,0,1,2,2,0,4,0,0,0,0,0,0,0,0]
On extrait 6	[3 (8/3/5)]	[-1,0,1,2,2,0,4,0,0,0,0,0,0,0,0]
On insère 8	[8 (8/6/2),3 (8/3/5)]	[-1,0,1,2,2,0,4,0,6,0,0,0,0,0,0]
On extrait 8	[3 (8/3/5)]	[-1,0,1,2,2,0,4,0,6,0,0,0,0,0,0]
On insère 11	[11 (8/7/1),3 (8/3/5)]	[-1,0,1,2,2,0,4,0,6,0,0,8,0,0]
On extrait 11	[3 (8/3/5)]	[-1,0,1,2,2,0,4,0,6,0,0,8,0,0]
On insère S	[S (8/8/0),3 (8/3/5)]	[-1,0,1,2,2,0,4,0,6,0,0,8,0,11]

On obtient comme tableau des prédécesseurs : [-1,0,1,2,2,0,4,0,6,0,0,8,0,11]. On s'arrête car on a marqué S dans le tableau des prédécesseurs. On remonte maintenant le tableau des prédécesseurs : 13 a pour prédécesseur 11 qui a pour prédécesseur 8 qui a pour prédécesseur 6 qui a pour prédécesseur 5 qui a pour prédécesseur 8 qui a pour prédécesseur 4 qui a pour prédécesseur 2 qui a pour prédécesseur 1 qui a pour prédécesseur E. Le chemin est donc E-1-2-4-6-7-8-11-S. Il ne reste plus qu'à le convertir en direction !

3.5 Bilan

Le choix de la structure de données était ici déterminant car la masse d'information à traiter était assez importante. Ainsi, le tas s'est révélé être un bon choix car plus optimal qu'une liste chainée classique (O(log n) conte O(n) pour les principales opérations). Les difficultés rencontrées furent surtout liées à la nécessité d'optimiser la moindre action pour pouvoir résoudre les exercices dans le temps imparti. Pour gagner plus de temps, on aurait pu implanter un tas de Fibonacci, structure plus complexe que le simple tas mais plus optimal encore. Sinon la traduction en graphe n'est pas forcement nécessaire, peut être qu'un gain de temps est possible sans passer par cette étape.
Tous les tests passent en un temps moyen total de 1,70 secondes chez moi.