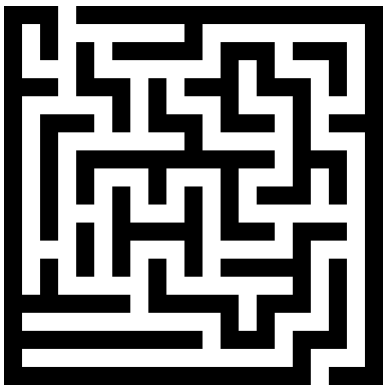
Notes de cours
Semaine 6

Cours Turing

1 Plus court chemin

Ces deux dernières semaines, nous avons vu comment générer des labyrinthes et comment y trouver son chemin. Nous avions vu que les labyrinthes en question étaient des instances d'un concept plus abstrait : celui des arbres. Pour rappel, un arbre est un ensemble de nœuds connectés de manière strictement hiérarchique. Dans ce genre de structures, il n'y a toujours qu'un unique chemin entre deux nœuds.

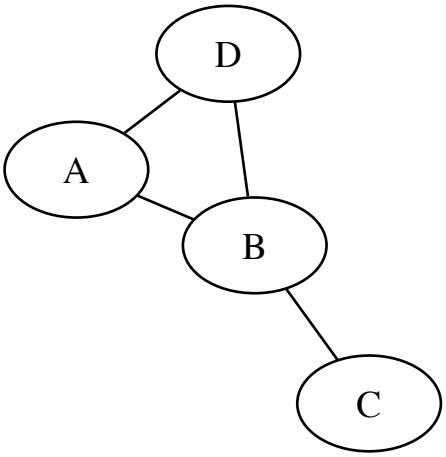
Pour cette dernière semaine sur les labyrinthes, nous allons considérer une généralisation de ce problème. Nous allons nous intéresser au cas où les labyrinthes ne forment plus simplement des arbres mais des structures plus générales. Par exemple, observez le labyrinthe ci-dessous et notez qu'il y a plus d'un chemin vers la sortie.



Cette semaine, nous allons étudier une méthode, l'algorithme A* (*A star*), pour non seulement trouver efficacement un chemin dans de telles structures, mais aussi garantir qu'il s'agit du chemin le plus court. Les labyrinthes sur lesquels nous allons travailler sont des exemples d'un concept plus général, celui des *graphes*.

1.1 Graphes

Tout comme un arbre, un graphe est une collection de nœuds reliés par des arêtes. Chaque arête relie exactement deux nœuds. Notez qu'il n'y a pas forcément d'arête entre chaque paire de nœuds. Voici un exemple de graphe.

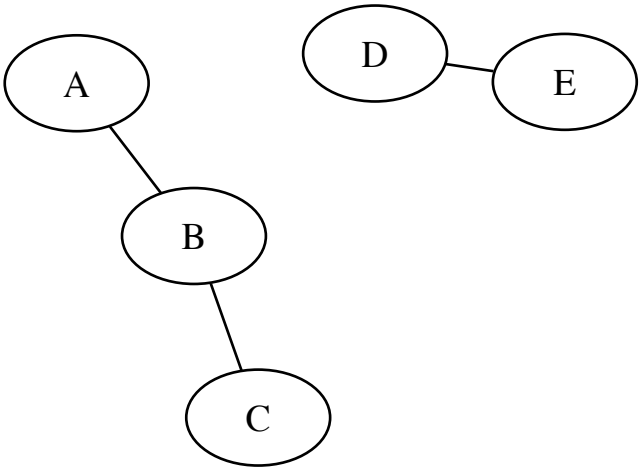


Remarquez que les arbres que nous avons vus la semaine dernière sont aussi des graphes. Les arbres sont juste des graphes avec des contraintes additionnelles.

Si l'on applique ce concept de graphes à des labyrinthes, chaque pièce du labyrinthe correspond à un nœud. On considère que deux nœuds sont reliés par une arête si les deux pièces sont voisines dans le labyrinthe.

1.2 Chemins

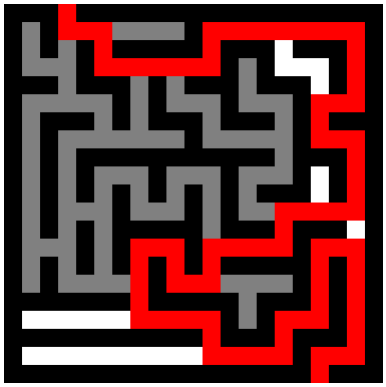
Un chemin dans un graphe est une séquence de nœuds où chaque paire de nœuds successifs sont reliés par une arête. Contrairement aux arbres, il n'y a pas toujours un unique chemin entre deux nœuds dans les graphes en général. Dans le graphe présenté plus haut, A - B - C est un premier chemin entre A et C, et A - D - B - C en est un deuxième. Notez que parfois, pour certains graphes, il n'y aura pas forcément de chemin entre chaque paire de nœuds, comme par exemple entre les nœuds A et E dans le graphe suivant.



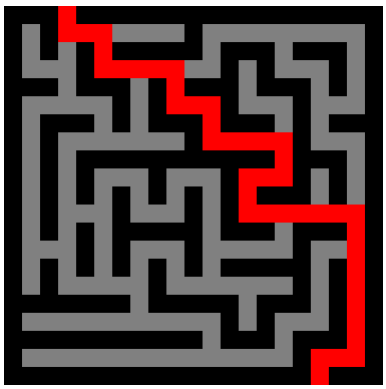
1.3 Comportement des méthodes de parcours

La semaine dernière, nous avons vu deux méthodes de parcours des arbres : le parcours en profondeur et le parcours en largeur. Ces méthodes de parcours, bien qu'abordées dans le cadre des arbres, peuvent être appliquées à des graphes plus généraux.

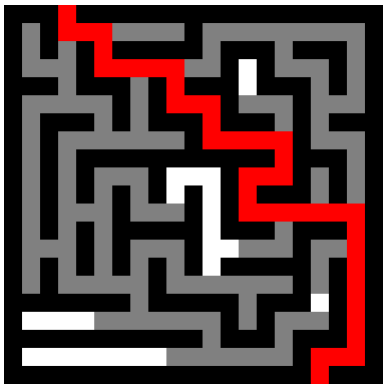
Parcours en profondeur La méthode de parcours en profondeur, qui explore les chemins jusqu'au bout et remonte en arrière en cas de cul-de-sac, peut s'appliquer à des graphes mais ne garantit pas de tomber sur le chemin le plus court. Il se peut que le chemin trouvé soit plus long que le plus court chemin possible. Ci-dessous par exemple est présenté le chemin trouvé par le parcours en profondeur sur le labyrinthe présenté plus tôt. Il s'agit bien d'un chemin de l'entrée à la sortie, mais le chemin n'est pas optimal.



Parcours en largeur La méthode d'exploration en largeur, quant à elle, garantit de retourner le chemin le plus court. Cependant, pour ce faire, la méthode requiert généralement d'explorer une grande partie du labyrinthe. Ci-dessous par exemple est présenté le chemin trouvé par le parcours en largeur sur le même labyrinthe. Notez que dans ce cas toutes les cases du labyrinthe ont dû être explorées avant d'arriver à ce chemin.



Afin de trouver un chemin efficacement, nous allons étudier ensemble un algorithme appelé A* (prononcé *A star*). L'algorithme A* permet de retrouver le plus court chemin dans un graphe de manière plus efficace que le parcours en largeur car il priorise l'exploration des nœuds sur des chemins prometteurs (c'est-à-dire des chemins courts et qui amènent proche de la sortie). Observez ci-dessous le résultat obtenu par cette méthode d'exploration.



Cette méthode nous permettra d'obtenir le plus court chemin en explorant moins du labyrinthe qu'avec un parcours en largeur. Notez que cette méthode requiert de pouvoir estimer si l'on s'approche ou l'on s'éloigne d'un objectif, par exemple en mesurant la distance qui sépare le nœud de la sortie sans tenir compte des murs.

L'algorithme A* est très similaire à la méthode de parcours en largeur et se base aussi sur l'utilisation d'une file. Cependant, l'algorithme A* utilise une *file de priorité* dans laquelle les éléments sont traités en fonction de leur niveau de priorité.

1.4 File de priorité

Tout comme les listes et les files, une file de priorité est une collection de valeurs. Une priorité est cependant associée à chaque valeur. Ces priorités indiquent l'ordre dans lequel les éléments doivent être traités. Plus la priorité d'un élément est basse, plus l'élément doit être traité rapidement. Les files de priorité supportent les deux opérations de base suivantes :

1. Ajout d'un élément dans la file à une priorité donnée.
2. Retrait de l'élément avec la priorité la plus basse.

Utilisation en Python La librairie standard de Python inclut un module pour les files, et notamment les files de priorité. Il s'utilise comme suit.

```
from queue import PriorityQueue  # Import de la classe

ma_file = PriorityQueue()  # Création d'une file de priorité

ma_file.put((1, "A"))  # Ajout d'un élément (priorité 1)
ma_file.put((0, "B"))  # Ajout d'un élément (priorité 0)
ma_file.put((4, "C"))  # Ajout d'un élément (priorité 4)

print(ma_file.get())  # Affiche (0, "B")
print(ma_file.get())  # Affiche (1, "A")
print(ma_file.get())  # Affiche (4, "C")
```

1.5 L'algorithme A*

L'algorithme A* est relativement simple. Il opère à l'aide d'une file de priorité des nœuds. L'algorithme s'exécute de façon itérative. À chaque itération, le nœud de plus faible priorité est retiré de la file et est ensuite traité. Pour traiter un nœud, on regarde premièrement s'il ne s'agit pas du nœud d'arrivée. Si c'est le cas, on reconstruit le chemin parcouru (comme pour le parcours en largeur de la semaine dernière) et on termine la recherche. Dans le cas où il ne s'agit pas du nœud d'arrivée, on insère dans la file tous les voisins du nœud qui soit n'ont jamais été atteints, soit sont atteints par un chemin plus court que précédemment. On note aussi la distance à laquelle le nœud peut être atteint depuis l'entrée et le nœud qui y mène (afin de pouvoir reconstruire le chemin à la fin). En terme de code, la structure de l'algorithme est la suivante :

```
def a_star(graphe, depart, arrivee):
    # File de priorité pour le traitement des noeuds
    file_noeuds = PriorityQueue()
    file_noeuds.put((0, depart))

    origines = {}  # Origine de chaque nœud atteint
    origines[depart] = None
    distances = {}  # Distances des noeuds atteints depuis le départ
    distances[depart] = 0

    while not file_noeuds.empty():
        # Récupération du prochain noeud
        _, noeud = file_noeuds.get()

        # Vérification de si on a atteint l'arrivée
        if noeud == arrivee:
            chemin = []
            while noeud is not None:
                chemin.append(noeud)
                noeud = origines[noeud]
            chemin.reverse()
            return chemin

        # Ajout des voisins s'ils sont plus rapidement atteignables
        for voisin in voisins(image, noeud):
            # Calcul de la distance du voisin au départ
            d = distances[noeud] + distance(graphe, noeud, voisin)
            if voisin not in distances or d < distances[voisin]:
                distances[voisin] = d
                origines[voisin] = noeud
                priorite = d + estimation(graphe, voisin, arrivee)
                file_noeuds.put((priorite, voisin))
```

Reste à déterminer les priorités auxquelles les nœuds sont insérés dans la file. La priorité de chaque voisin consiste en la somme de deux valeurs :

1. Le distance du nœud depuis le nœud de départ. Cette distance peut être calculée facilement en maintenant un dictionnaire avec la distance effective depuis le départ pour chaque nœud inséré dans la file. Au moment de l'insertion dans la file, il est facile de calculer cette valeur.
2. Une estimation de la distance qui sépare le voisin de l'arrivée. Pour que le chemin retourné soit le plus court, il faut s'assurer que cette estimation ne surestime jamais la véritable distance. [1] On appelle ce genre de fonctions des *heuristiques*. Plus bas sont listées différentes heuristiques applicables dans le cadre de cet algorithme, plus ou moins appropriées selon le contexte.

Notez que dans le cas où l'heuristique choisie indique toujours 0 comme estimation de la distance au nœud d'arrivée, on retombe sur l'algorithme de parcours en largeur.

1.5.1 Distance de Manhattan

Dans notre cas, comme on peut uniquement se déplacer à gauche, à droite, en haut ou en bas, il est relativement simple d'estimer le nombre minimal d'étapes à parcourir dans le meilleur des cas. Il s'agit simplement de la distance horizontale additionnée à la distance verticale.

$$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

On appelle cette distance la *distance de Manhattan*, en référence à l'organisation quadrillée de cet arrondissement de New York. La distance de Manhattan constitue une bonne heuristique dans le cas de nos labyrinthes. En effet, elle ne surestime jamais la distance réelle et y est même égale dans certains cas.

1.5.2 Autres heuristiques

Notez qu'il existe d'autres distances que l'on peut utiliser comme heuristiques, c'est-à-dire comme approximations de la distance réelle. Par exemple, la distance euclidienne, qui permet de mesurer la distance à vol d'oiseau, est une heuristique adéquate.

$$d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Il y a aussi des distances intéressantes dans le cas de déplacements dans une grille avec des déplacements en diagonale. Par exemple, la distance de Tchebychev, qui mesure la plus grande des distances sur un seul axe et qui permet de mesurer le nombre de cases à traverser si les sauts en diagonale sont autorisés (et comptent pour une distance de 1).

$$d((x_1, y_1), (x_2, y_2)) = \max(|x_1 - x_2|, |y_1 - y_2|)$$

1. Pour être exact, afin de s'assurer de ne pas devoir visiter à répétition un même nœud, il faut aussi que la fonction soit *monotone* : pour chaque nœud, la distance estimée à l'arrivée doit être plus petite ou égale à la somme de la distance effective à un voisin additionnée à la distance estimée du voisin à l'arrivée. Les fonctions proposées ont toutes cette propriété.

Dans le cas où l'on autorise les sauts en diagonale mais que l'on souhaite les comptabiliser de manière géométrique, on utilisera la distance suivante :

$$d((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2| + (\sqrt{2} - 2) \cdot \min(|x_1 - x_2|, |y_1 - y_2|)$$

Le choix de la distance, plus ou moins précise, influencera le nombre de nœuds qui seront visités par l'algorithme A*. Toutefois, le choix de la distance n'a pas d'influence sur l'optimalité du résultat.

2 Idées d'extensions

Une fois l'algorithme A* implementé dans le cadre des labyrinthes, il vous sera en principe simple de l'appliquer à d'autres contextes. Dans cette section, nous explorerons deux de ces différents contextes :

1. Le plus court chemin sur une carte avec des obstacles et la possibilité de se déplacer en diagonale.
2. Le plus court chemin sur une carte avec des cases à différentes hauteurs.

2.1 Première extension

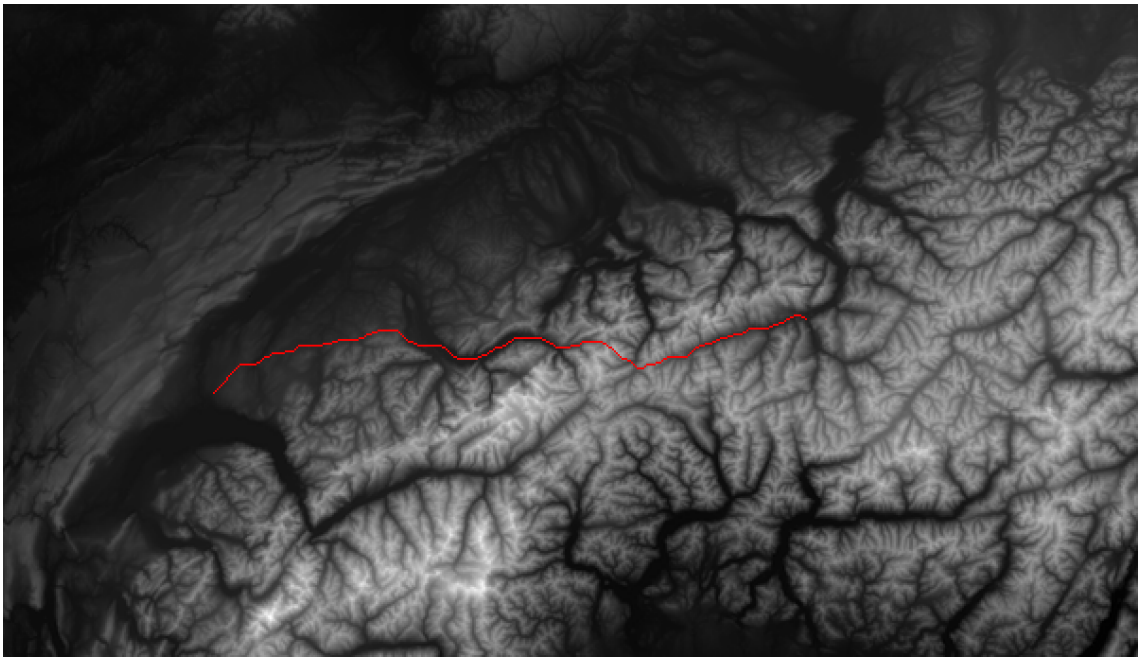
Pour la première extension, concevez un programme qui prend en entrée une image et qui y recherche un chemin en évitant les zones infranchissables. Les pixels blancs indiquent des zones qu'il est possible de visiter, les pixels noirs des zones infranchissables. Dans cette image, il faudra trouver un pixel rouge (le nœud de départ) et un pixel vert (le nœud d'arrivée). Trouvez ensuite le chemin le plus court entre ceux deux nœuds. Notez en gris les nœuds visités.

Pour cet exercice, nous considérerons qu'il est possible de se déplacer à gauche, à droite, en haut et en bas, mais aussi en diagonale. Alors que les déplacements verticaux et horizontaux auront un coût de 1 unité, les déplacements en diagonale auront un coût de $\sqrt{2}$.

Vous trouverez des images d'exemple sur Moodle. Vous êtes aussi libres de créer vos propres cartes et d'apporter d'autres améliorations ! On pourrait par exemple imaginer avoir des zones de couleurs différentes dans lesquelles on progresse plus vite ou plus lentement. Par exemple, on pourrait représenter des marais en vert et des routes pavées en jaune. Il serait alors logique qu'un déplacement par la route prenne moins de temps et qu'un déplacement au travers d'un marais en prenne plus.

2.2 Deuxième extension

Pour cette deuxième extension, faites en sorte de trouver le plus court chemin sur une carte d'élévation. En entrée, votre programme prendra une image sur laquelle l'élévation sera indiquée par un niveau de gris. Plus une zone sera élevée, plus elle sera proche du blanc. Au contraire, plus une zone est basse, plus elle sera proche du noir. Dans le cadre de cet exercice, on considérera que la somme des trois composantes de la couleur (le rouge, le vert et le bleu) d'un pixel représente la hauteur d'une case.



Faites en sorte de demander à l'utilisateur la position du point de départ et du point d'arrivée dans l'image et de retourner le chemin avec le plus petit coût du départ à l'arrivée. Le coût d'un chemin sera comptabilisé comme ceci :

1. 1 pour un déplacement d'une case à l'horizontale ou à la verticale sur le chemin.
2. $\sqrt{2}$ pour un déplacement d'une case en diagonale sur le chemin.
3. $0,05$ par unité de hauteur de différence d'une case à l'autre sur le chemin.

Comme pour l'extension précédente, vous trouverez des images d'exemple sur Moodle. Vous êtes aussi libre de créer vos propres cartes à l'aide de l'outil `https://tangrams.github.io/heightmapper` par exemple.