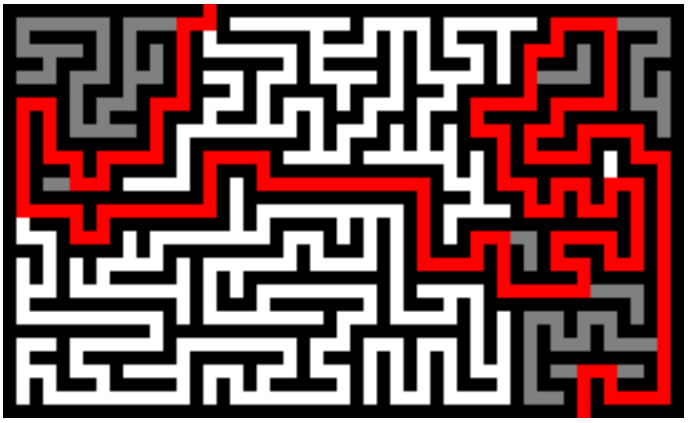
Notes de cours
Semaine 8

Cours Turing

1 Résolution de labyrinthes

La semaine dernière nous avons vu comment générer des labyrinthes sous forme d'images en Python. Cette semaine nous allons voir, sous la forme d'un projet, comment résoudre ces mêmes labyrinthes, c'est-à-dire trouver un chemin de l'entrée à la sortie du labyrinthe.

Ci-dessous est présenté un exemple de ce que l'on cherche à obtenir aujourd'hui. Le chemin de l'entrée à la sortie est affiché en rouge, les zones explorées mais non retenues en gris.



Pour cela, nous allons voir deux approches de résolution :

1. le parcours en profondeur, et
2. le parcours en largeur.

Après ces deux approches, nous aborderons des généralisations de ce problème dans le cadre des *graphes* et des algorithmes de plus court chemin.

1.1 Bases communes

Avant de plonger dans la découverte des différentes approches, il nous faut poser quelques bases communes aux deux approches.

1.1.1 Nœud

Tout d'abord, intéressons-nous au concept de nœud. La semaine dernière, nous avions le concept de pièces reliées par des couloirs, avec une distinction entre les pièces et les couloirs. Dans l'étape du jour, nous n'aurons pas besoin de faire cette différence, et pour cela nous utiliserons juste le concept de *nœud*.

Les différentes positions accessibles dans le labyrinthe seront des nœuds, *reliés entre eux* si l'on peut se déplacer directement de l'un à l'autre.

Quant aux murs, ils ne seront pas représentés par des nœuds. Deux nœuds séparés par un mur ne seront pas reliés entre eux.

En termes d'implémentation, chaque nœud correspondra à exactement un pixel de l'image du labyrinthe. Au départ, tous les nœuds du labyrinthe seront représentés par les pixels blancs de l'image. On identifiera donc un nœud grâce à ses coordonnées x et y.

1.1.2 Couleur des nœuds

La couleur du pixel associé à un nœud nous servira à indiquer si le nœud a déjà été visité ou encore s'il fait partie du chemin jusqu'à la sortie. Nous utiliserons un pixel gris pour indiquer que le nœud a été visité lors de la recherche mais qu'il ne mène pas à la sortie. Au final, nous utiliserons la couleur rouge pour indiquer les nœuds sur le chemin vers la sortie. La couleur blanche sera réservée aux nœuds qui n'ont pas été visités.

```
def est_libre(image, noeud):
    return image.getpixel(noeud) == (255, 255, 255)
```

1.1.3 Voisins et voisins non-visités

On appelle les nœuds situés directement au-dessus, au-dessous, à gauche ou à droite d'un nœud ses voisins. Notez que, contrairement à l'étape précédente, les voisins sont directement adjacents et ne sont pas séparés par un couloir.

Nos deux méthodes de résolution auront besoin d'une fonction pour déterminer les voisins *non-visités* d'un nœud. Les voisins non-visités sont représentés par des pixels blancs dans l'image du labyrinthe. Il sera donc intéressant d'implémenter une fonction pour donner la liste des voisins non-visités d'un nœud. Attention à la gestion des bordures de l'image.

```
def voisins_non_visites(image, noeud):
    voisins = []
    # À compléter
```

1.1.4 Nœud de départ

Dans les labyrinthes que l'on cherchera à résoudre, il y aura toujours un unique nœud de départ situé sur le haut du labyrinthe. Il sera intéressant d'implémenter une fonction pour trouver les coordonnées x et y du nœud de départ à partir de l'image du labyrinthe.

```
def noeud_depart(image):
    # À compléter
```

1.1.5 Dessiner un chemin

Lorsque nous aurons trouvé un chemin du départ à l'arrivée dans le labyrinthe, il faudra indiquer ce chemin en rouge dans l'image. Pour cela, vous pourrez utiliser la fonction suivante :

```
def dessiner_chemin(image, chemin):
    for noeud in chemin:
        image.putpixel(noeud, (255, 0, 0))
```

1.2 Parcours en profondeur

Le parcours en profondeur (*depth-first search* en anglais) est une méthode de parcours des labyrinthes et plus généralement de parcours dans ce que l'on appelle des *arbres*. Nous aurons l'occasion de parler des arbres en plus de détails tout prochainement dans la suite du cours.

Le parcours en profondeur est une méthode qui consiste à explorer les chemins jusqu'au bout avant de tenter d'explorer d'autres chemins. À chaque étape d'un chemin, à chaque nœud, on choisit arbitrairement un nœud voisin non-visité pour s'y déplacer. Au bout du chemin, soit on arrive à la sortie et on arrête le parcours, soit on tombe sur un cul-de-sac. Dans le cas d'un cul-de-sac, on rebrousse chemin jusqu'au dernier nœud avec des voisins encore non-visités. Notez que cette méthode d'exploration des labyrinthes est très similaire à celle employée la semaine dernière pour générer le labyrinthe.

En termes d'implémentation en Python, il vous faudra implémenter une fonction nommée `parcours_en_profondeur` qui prend deux arguments :

1. l'image du labyrinthe, et
2. la position de départ.

La fonction devra retourner, sous la forme d'une liste, les positions des nœuds sur le chemin du départ à l'arrivée. Les positions de départ et d'arrivée devront être contenues dans cette liste. L'image pourra être modifiée par la fonction, pour, par exemple, indiquer quel nœuds ont été visités.

```
def parcours_en_profondeur(image, pos_depart):
    x, y = pos_depart
    chemin = []
    # À compléter
    return chemin
```

Il est possible de l'implémenter à l'aide d'une boucle, auquel cas il faudra maintenir une liste Python pour contenir le chemin parcouru depuis le noeud de départ jusqu'à la cellule courante. Il est aussi possible d'implémenter cette fonction de manière récursive. Une fois le chemin retourné, vous pourrez utiliser la fonction `dessiner_chemin` afin de colorier ce chemin en rouge.

1.3 Parcours en largeur

Le parcours en largeur *breadth-first search* est une autre méthode de parcours qui explore les noeuds à des distances de plus en plus grandes du nœud de départ. Avant d'explorer un nœud à une distance $d+1$ de la position de départ, tous les nœuds à distance d doivent avoir été explorés.

Les files Afin de réaliser un parcours en profondeur, on s'aide généralement d'une *file* (*queue* en anglais). Une file est une structure de données, une collection de valeurs, qui permet de réaliser efficacement deux opérations de base :

1. L'ajout d'un ou plusieurs éléments en fin de file.
2. Le retrait d'un élément en début de file.

Les files sont des structures de données dites FIFO (pour *first-in first-out*, littéralement *premier dedans, premier dehors*). Elles permettent de traiter les éléments dans l'ordre où ils sont insérés, comme une file d'attente au magasin ou à la cafétéria.

Notez que, à contrario, les listes Python permettent de faire efficacement des traitements dans l'ordre LIFO (pour *last-in, first-out*, *dernier dedans, premier dehors*). En effet, pour les listes Python, l'ajout et le retrait à la fin de la liste sont efficacement effectués. Pour les files, on peut ajouter efficacement à la fin et retirer efficacement au début.

En Python, il est possible d'utiliser très simplement les files à l'aide du module `collections`. Le module exporte une collection appelée `deque` (pour *double-ended queue*). Cette collection représente une file améliorée qui permet l'ajout et le retrait efficace des deux côtés de la file.

```
from collections import deque

ma_file = deque()  # Création d'une file
ma_file.append(1)  # Ajout d'un élément
ma_file.extend([2, 3, 4])  # Ajout de plusieurs éléments
elem = ma_file.popleft()  # Retrait du premier élément
```

Description du parcours en profondeur Le parcours en profondeur se base sur l'utilisation d'une file de nœuds. Initialement, la file ne contiendra que le nœud de départ. Puis, de manière répétée et tant que la file n'est pas vide, on retirera le premier élément de la file, on vérifiera que cet élément n'est pas le nœud d'arrivée, puis on ajoutera en fin de file tous les voisins non-visités du nœud. Si l'on tombe sur le nœud d'arrivée, le parcours se termine.

Cette manière de fonctionner, très simple, ne permet cependant pas directement de reconstruire un chemin du départ à l'arrivée. En effet, on perd trace du chemin qui mène à un nœud une fois ce nœud ajouté à la file. Pour reconstruire le chemin, il nous faudra nous aider d'une

autre structure de données. Cette structure additionnelle nous permettra de noter, pour chaque nœud visité (autre que le nœud de départ), le nœud par lequel on était arrivé. Pour cela, la collection la plus adaptée sera un dictionnaire.

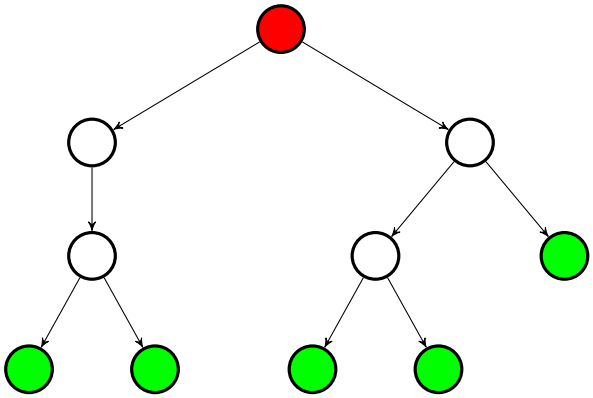
À chaque fois que l'on ajoute un nœud à la file, on ajoutera aussi au dictionnaire le nœud par lequel on y accède. Une fois le nœud d'arrivée trouvé, il suffira de consulter ce dictionnaire afin de remonter le chemin inverse.

Comparaisons avec la recherche en profondeur La recherche en profondeur et la recherche en largeur sont deux approches très similaires. D'ailleurs, en changeant un unique appel de méthode, vous devriez être capable de transformer votre implémentation de la recherche en largeur en une recherche en profondeur !

La recherche en largeur a cependant l'avantage de retourner le plus court chemin même en présence de boucles dans les labyrinthes. Ce cas de figure ne se présente pas encore, mais nous l'étudierons la semaine prochaine !

2 Les arbres

Les labyrinthes que vous avez générés la semaine passée et que nous avons résolus aujourd'hui sont des exemples d'un concept plus abstrait que l'on appelle les *arbres*. Un arbre est une structure de données formée de nœuds connectés de manière hiérarchique. Voici la représentation graphique d'un arbre.



Il existe un vocabulaire assez riche pour parler des différents éléments d'un arbre. Ci-dessous, nous allons présenter brièvement quelques-uns de ces termes.

Nœuds Les arbres sont composés de *nœuds*. Dans le schéma ci-dessus, ils sont représentés par des cercles. Dans le cas des labyrinthes, les nœuds de l'arbre correspondent aux différentes zones où il est possible de se déplacer.

Racine L'arbre a une unique *racine* (en rouge dans le schéma), un unique nœud qui est le point d'entrée dans l'arbre. Dans le cas des labyrinthes, il s'agissait du nœud de départ.

Arêtes Les nœuds d'un arbre peuvent être reliés à d'autres nœuds par le biais d'une *arête*. Les arêtes vont toujours d'un nœud à un autre nœud plus bas dans l'arbre, et ainsi ne forment jamais de *cycles* (de boucles).

Parent et enfants On appelle les nœuds directement en dessous d'un nœud et reliés à celui-ci par une arête les *enfants* du nœud. Selon la même métaphore, on appelle le *parent* d'un nœud le nœud dont il est l'enfant. Chaque nœud, à part la racine, a un unique parent.

Feuilles Certains nœuds n'ont pas d'enfants, on appelle ces nœuds des *feuilles* (en vert dans le schéma). Dans le contexte des labyrinthes, le nœud à la sortie est une feuille, tout comme les nœuds qui sont dans un cul-de-sac.

Étiquettes Les arbres peuvent aussi contenir des valeurs. Ces valeurs peuvent être liées aux nœuds comme aux arêtes. On appelle ces valeurs liées aux éléments de l'arbre des *étiquettes*.

Applications des arbres

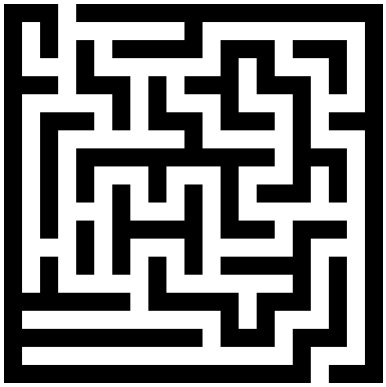
Les arbres ont de nombreuses applications en informatique. Ils permettent de représenter des données hiérarchiques comme l'on retrouve par exemple dans les systèmes de fichiers (avec dossiers, sous-dossiers, etc.) ou l'organisation de grandes entreprises. Les arbres sont aussi au cœur d'algorithmes, comme par exemple l'algorithme de Huffman pour la compression de données. Ils se retrouvent aussi dans le domaine de l'intelligence artificielle sous la forme d'arbres de décision. Comme nous le verrons aussi à la toute fin du cours, les arbres peuvent aussi servir à représenter les expressions d'un langage de programmation et même la structure d'un programme.

Les deux méthodes de parcours que l'on a vues aujourd'hui sont applicables à n'importe quel arbre, et même à des structures plus générales appelées *graphes*.

3 Plus court chemin

Jusqu'à présent, nous avons vu comment générer des labyrinthes et comment y trouver son chemin. Nous venons de voir que les labyrinthes en question sont des instances d'un concept plus abstrait : celui des arbres. 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.

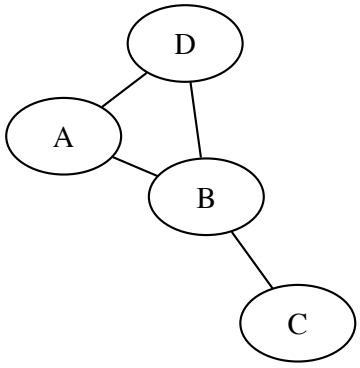
Pour cette dernière partie du module, nous allons considérer une généralisation de ce problème. Nous allons nous intéresser au cas où les labyrinthes ne forment plus simplement des arbres mais des structures plus générales. Par exemple, observez le labyrinthe ci-dessous et notez qu'il y a plus d'un chemin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



Pour cela, 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*. Cet algorithme A* est très utilisé en pratique, par exemple dans les jeux vidéo pour permettre aux personnages non-joueurs de se déplacer efficacement dans leur environnement.

3.1 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.

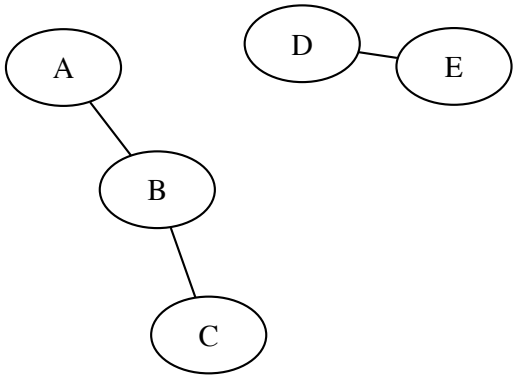


Remarquez que les arbres que nous avons vus la semaine dernière sont aussi des graphes. Les arbres sont juste des graphes avec des contraintes additionnelles.

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.

3.2 Chemins

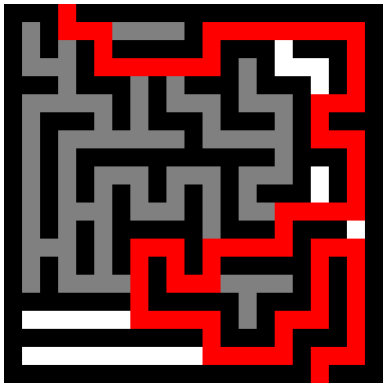
Un chemin dans un graphe est une séquence de nœuds où chaque paire de nœuds successifs sont reliés par une arête. Contrairement aux arbres, il n'y a pas toujours un unique chemin entre deux nœuds dans les graphes en général. Dans le graphe présenté plus haut, A - B - C est un premier chemin entre A et 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.



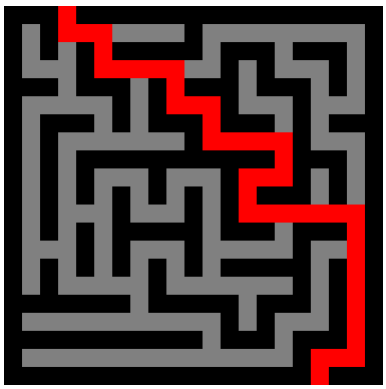
3.3 Comportement des méthodes de parcours

Jusqu'à présent, 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.

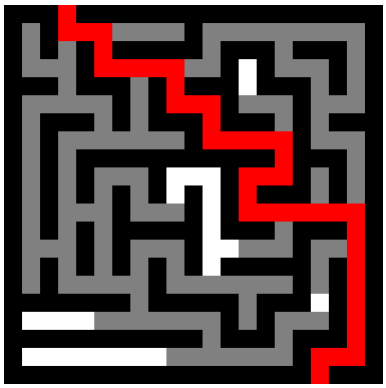
Parcours en profondeur La méthode de parcours en profondeur, qui explore les chemins jusqu'au bout et remonte en arrière en cas de cul-de-sac, peut s'appliquer à des graphes mais ne garantit pas de tomber sur le chemin le plus court. Il se peut que le chemin trouvé soit plus long que le plus court chemin 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.

3.4 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 valeur de priorité donnée.
2. Retrait de l'élément le plus prioritaire. Contre-intuitivement, l'élément le plus prioritaire est celui avec *la plus faible* valeur de priorité.

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")
```

3.5 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.

3.5.1 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.

3.5.2 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.

4 Idées d'extensions

Une fois l'algorithme A* implémenté 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.

4.1 Première extension

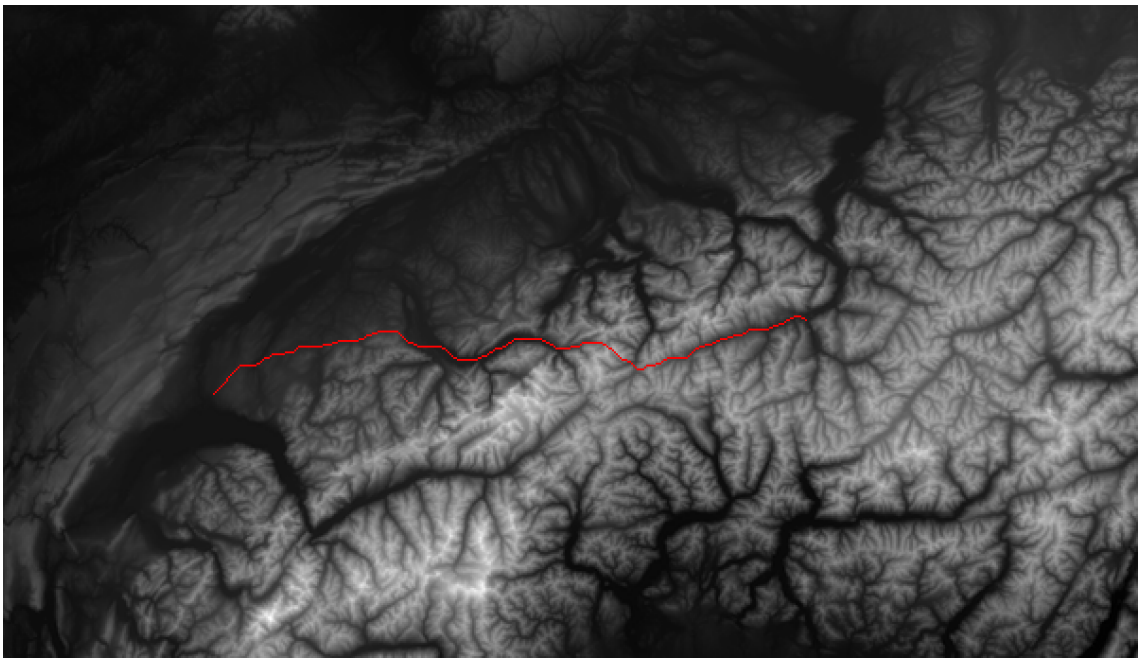
Pour la première extension, concevez un programme qui prend en entrée une image et qui y recherche un chemin en évitant les zones infranchissables. Les pixels blancs indiquent des zones qu'il est possible de visiter, les pixels noirs des zones infranchissables. Dans cette image, il faudra trouver un 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.

4.2 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.