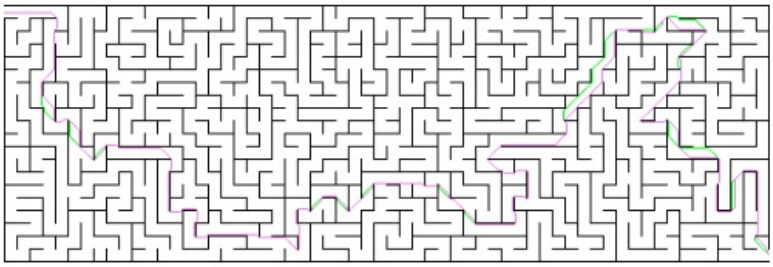
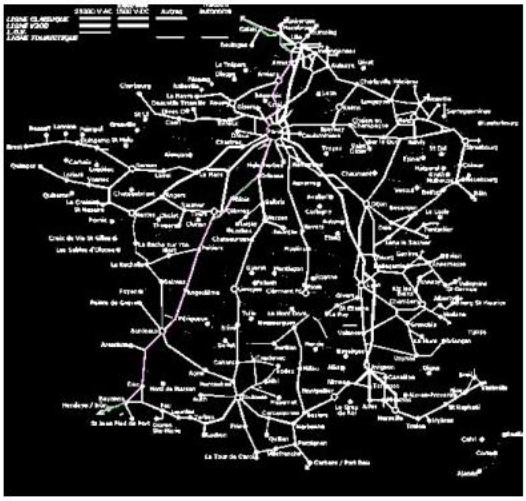
TP6 – BASES DES GRAPHES ALGORITHME A* (ET DIJKSTRA)

Contexte

On souhaite trouver le plus court chemin à réaliser sur une image en noir et blanc entre un départ et une arrivée en ne considérant que les mouvements possibles ←↙↓↘→↗↑↖. Les pixels noirs seront des obstacles, les blancs des cases accessibles. Voici des exemples de chemins que vous pourrez trouver dans la dernière partie de ce TP :





Nous allons donc :

- Importer des fonctions d'affichage prédéfinies
- Réaliser la grille/image de départ
- Créer un dictionnaire représentant le graphe du domaine d'étude
- Trouver le plus court chemin à l'aide de l'algorithme de Dijkstra
- Faire de même avec l'algorithme A-star
- Comparer performances de ces algorithmes

Pour rappel :

- L'algorithme de Dijkstra permet de trouver le plus court chemin dans un graphe en « avançant » par « arcs de cercles » depuis le départ, jusqu'à atteindre l'arrivée.
- L'algorithme A-star privilégie dans ses choix, les cases dont l'heuristique (distance depuis le début + distance à vol d'oiseau jusqu'à l'arrivée) est la plus faible. Dans ce TD, il cherchera donc toujours à se diriger vers l'arrivée.

Un peu d'histoire

Les premiers algorithmes de recherche de plus court chemin comme le parcours en largeur (*Breadth First Search* en anglais), le parcours en profondeur (*Depth First Search* en anglais) ainsi que l'algorithme de Dijkstra ont été largement utilisés jusqu'à ce que l'algorithme A* proposé par Hart, Nilsson et Raphael en 1967 montre un grand potentiel dans le cadre des applications aux jeux vidéos [1].

Ce TP se focalise sur l'algorithme A* bien que des variantes existent comme l'algorithme IDA* (*Iterative Deepening A**) ou REA* (*Rectangle Expansion A**).

Principe de l'algorithme A*

Contrairement à l'algorithme de Dijkstra qui réalise une recherche du plus court chemin en parcourant une grande portion du graphe, l'algorithme A* utilise une méthode qui réduit l'espace de recherche en ne considérant que le nœuds les plus prometteurs. Les nœuds les plus prometteurs sont déterminés en donnant de l'information supplémentaire dans l'algorithme : la recherche est alors dite *heuristique* ou *informée*.

L'illustration donnée en 3 et issue de l'article de Cui et Shi [2] montre la différence entre ces 2 stratégies de recherche du plus court chemin. Les cases violettes sont celles qui ont été considérées dans le cadre de la recherche du chemin optimal. L'algorithme A* en compte beaucoup moins que l'algorithme de Dijkstra, il est légitime de penser que l'algorithme A* sera plus rapide. Ceci est possible grâce à l'heuristique choisie.

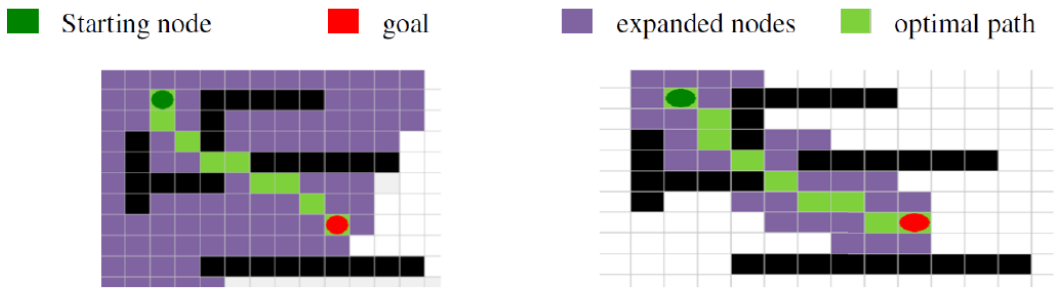


FIGURE 3 – Comparaison des algorithmes de Dijkstra et A*. L'algorithme de Dijkstra envisage beaucoup plus de points à visiter.

Pour répondre à la problématique posée, l'algorithme A* sera implémenté et plusieurs heuristiques seront testées afin d'obtenir le chemin optimal entre 2 points le plus vite possible.

En Annexe vous avez différentes comparaisons d'algorithmes qui ont été réalisé avec le très bon site https://qiao.github.io/PathFinding.js/visual/ sur la carte suivante :

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	C20	C21	C22	C23	C24	C25	C26	
L1	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	
L2	1	1	1	1	0	0	1	1	1	0	0	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	
L3	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	
L4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
L5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
L6	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	
L7	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
L8	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
L9	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	1	1	0	0	
L10	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	
L11	1	1	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
L12	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
L13	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
L14	1	1	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
L15	1	1	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

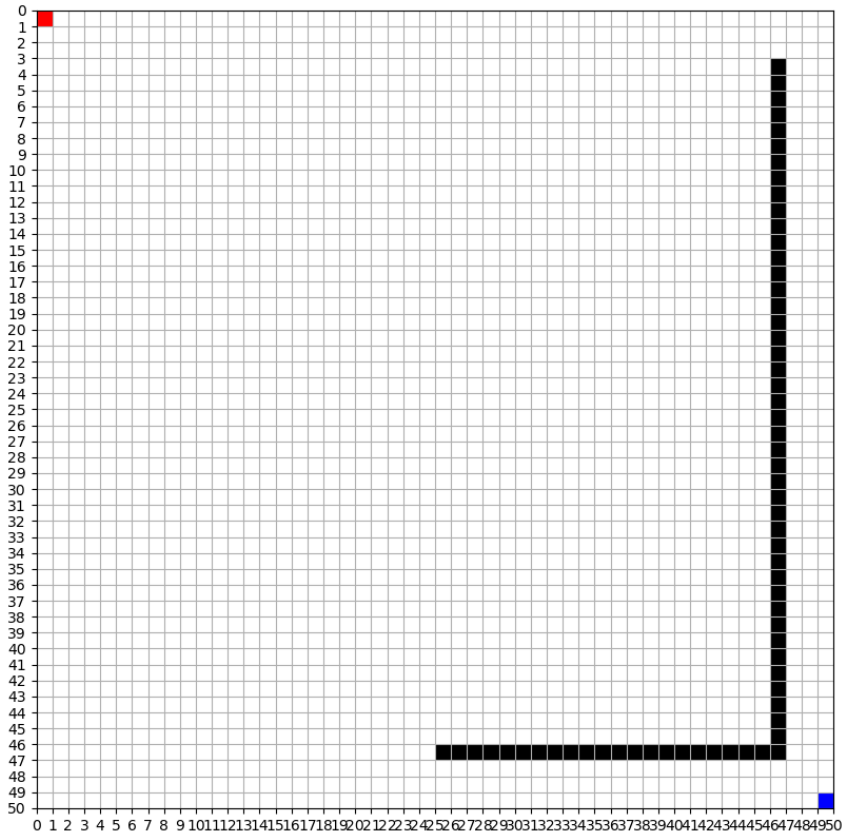
Prise en main du code élèves

Télécharger le dossier contenant tous les fichiers et images… Pensez à dézipper l'archive afin d'avoir le dossier voulu !
Vous avez à disposition 5 fichiers Python qui vont être utilisés ici :

11-3 - 1 - Affichage	Import des librairies numpy et matplotlib Création de 3 fonctions d'affichage : Affiche(fig,im,grille) : Affiche l'image d'étude Affiche_Save(fig,im,grille,chemin) : Enregistre l'image pour créer des animations (sans affichage sinon bug de redimensionnement) Affiche_Degrade(Fig,Tab) : Affiche avec un dégradé les distances de chaque pixel depuis le départ sur une nouvelle image
11-3 - 2 - Grille	Code pour créer la grille de départ
11-3 - 3 - Dico - Elèves	A compléter
11-3 - 4 - Dijkstra – Elèves	A compléter
11-3 - 5 - A star - Elèves	A compléter

Question 1: Télécharger et exécuter les fichiers Affichage et Grille dans l'ordre

A ce stade, vous devriez voir apparaître le domaine d'étude sous la forme suivante :



La grille est remplie de cases blanches (accessibles), on voit apparaître le point de départ en rouge, le point d'arrivée en bleu et les cases obstacles en noir.

Les autres fichiers sont des images supplémentaires (dernière partie) ou les corrections des fichiers Dico, Dijkstra et Astar.

Réalisation du dictionnaire des voisins

Les mouvements possibles pour se déplacer d'une case à l'autre sont les 8 directions :

- ↑→↓← de distance 1
- ↖↗↘↙ de distance $\sqrt{2}$

On souhaite réaliser un dictionnaire nommé « Dico_Voisins » représentant le graphe du système tel que :

- Les clés sont des tuples de type (ligne,colonne) de chaque pixel pour chaque case/pixel accessible (pas noire). Exemple d'une portion des clés du dictionnaire :

```
>>> Dico_Voisins.keys()
dict_keys([(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8
), (0, 9), (0, 10), (0, 11), (0, 12), (0, 13), (0, 14), (0, 15), (0, 16), (0, 17
), (0, 18), (0, 19), (0, 20), (0, 21), (0, 22), (0, 23), (0, 24), (0, 25), (0, 2
6), (0, 27), (0, 28), (0, 29), (0, 30), (0, 31), (0, 32), (0, 33), (0, 34), (0,
35), (0, 36), (0, 37), (0, 38), (0, 39), (0, 40), (0, 41), (0, 42), (0, 43), (0,
```

- Les valeurs associées à chaque clé/case sont des listes contenant des sous listes (une par case voisine accessible) de deux éléments du type [clé case,distance]. Exemple :

```
>>> Dico_Voisins[(0,0)]
[[(0, 1), 1.0], [(1, 0), 1.0], [(1, 1), 1.4142135623730951]]
```

Vous travaillerez dans le fichier nommé « 11-3 - 3 - Dico - Elèves.py ».

Question 2: Créer une fonction Test_Pix(P1,P2) qui renvoie le booléen répondant à la question « Le pixel P1 est égal au pixel P2 »

Vérifier :

```
>>> Test_Pix([255,255,255],[0,0,0])
False

>>> Test_Pix([0,0,0],[0,0,0])
True
```

```
>>> Test_Pix(Image[1,1],[255,255,255])
True

>>> Test_Pix(Image[1,1],[0,0,0])
False
```

Question 3: Mettre en place le code permettant de créer le dictionnaire Dico_Voisins contenant une liste vide pour chaque case accessible

Remarque : Penser qu'une case accessible est « non noire », et non… blanche.

Vous devriez voir :

```
>>> len(Dico_Voisins)
2435

>>> Dico_Voisins[(0,0)]
[]
```

Question 4: Mettre en place la fonction Couples_Voisins(l,c) qui, pour la case à la ligne l et la colonne c supposée accessible, ajoute dans sa liste dans Dico_Voisins, les couples [Case_i,Di] des cases voisins accessibles (max 8 cases possibles)

Remarque :

- On évitera le test « in Dico_Voisins » pour vérifier qu'une case est accessible, qui coûte bien plus cher que de vérifier la couleur non noire d'un pixel
- Penser que sur tous les bords de l'image, il n'y a plus 8 voisins… Cela se traduira par des conditions sur li et ci des cases voisines « Case_i »

Vérifier :

```
>>> Couples_Voisins(0,0)

>>> Dico_Voisins[(0,0)]
[[(0, 1), 1.0], [(1, 0), 1.0], [(1, 1), 1.4142135623730951]]
```

Question 5: Mettre en place les lignes de code permettant de remplir Dico_Voisins (pour cases accessibles uniquement)

Vérifier :

```
>>> Dico_Voisins[(0,0)]
[[(0, 1), 1.0], [(1, 0), 1.0], [(1, 1), 1.4142135623730951]]

>>> Dico_Voisins[(49,49)]
[[(48, 48), 1.4142135623730951], [(48, 49), 1.0], [(49, 48), 1.0]]

>>> Dico_Voisins[(43,43)]
[[(42, 42), 1.4142135623730951], [(42, 43), 1.0], [(42, 44), 1.4142135623730951], [(43, 42), 1.0],
[(43, 44), 1.0], [(44, 42), 1.4142135623730951], [(44, 43), 1.0], [(44, 44), 1.4142135623730951]]

>>> Dico_Voisins[(42,42)]
[[(41, 41), 1.4142135623730951], [(41, 42), 1.0], [(41, 43), 1.4142135623730951], [(42, 41), 1.0],
[(42, 43), 1.0], [(43, 41), 1.4142135623730951], [(43, 42), 1.0], [(43, 43), 1.4142135623730951]]

>>> Dico_Voisins[46,46]
Traceback (most recent call last):
  File "<console>", line 1, in <module>
KeyError: (46, 46)
```

Mise en place de l'algorithme de Dijkstra

Vous travaillerez dans le fichier nommé « 11-3 - 4 - Dijkstra - Elèves.py ».

Ce fichier contient un paragraphe d'initialisation :

Distances	Array aux mêmes dimensions que l'image traitée, tel que Distances[l,c] est la distance du chemin menant à la case concernée depuis le départ Il est initialisé avec des valeurs infinies
Reste	Copie du dictionnaire Dico_Voisins. On enlèvera les cases traitées de ce dictionnaire et l'algorithme se terminera au plus tard, quand ce dictionnaire est vide
Provenances	Dictionnaire des provenances de chaque case. Initialisé vide, il contiendra à la fin de la réalisation de l'algorithme des clés et valeurs qui seront respectivement le nom de la case concernée et la case qui a permis d'y accéder lors de l'exécution de l'algorithme. Par exemple, si Provenances[(1,1)]=(0,0), c'est que pour atteindre la case (1,1) avec un chemin depuis le départ de longueur Distances[1,1], la case précédente de ce chemin est (0,0)

Pour rappel, départ et arrivée ont été définies précédemment (ld,cd,la,ca).

Nous allons maintenant programmer l'algorithme de Dijkstra, dont nous allons rappeler les grandes étapes.

Tant que **S** n'est pas l'arrivée, qu'il reste des stations dans **Reste** et que Distance[lS,cS] est différente de l'infini (cette condition permet de gagner du temps si **Arrivee** n'est pas accessible depuis **Depart**)

- **S** ← Station parmi **Reste** ayant la distance minimum dans **Distances**
- Mise à jour de **Reste** (retirer S)
- **Voisins** ← Dictionnaire des stations de **Reste** voisines de **S**
- Traitement des voisins : Pour chaque voisin V, si la distance Depart→S→V < Depart→V actuellement stockée dans **Distances** :
    - Mise à jour de **Distances** avec cette nouvelle distance Depart→S→V
    - Mise à jour de **Provenances** afin d'indiquer que S est le prédécesseur de V

Question 6: Mettre en place cet algorithme

On remarquera que si la case arrivée n'est pas accessible, Reste sera vide, mais en plus, Distances[la,ca] sera toujours infinie.

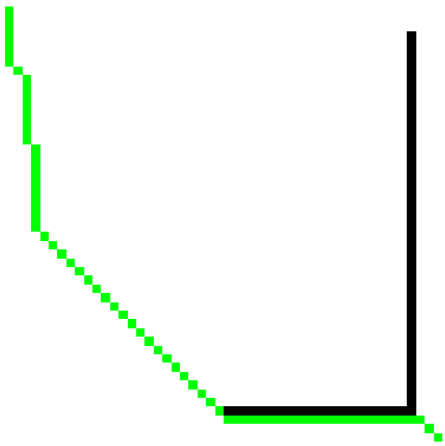
Si vous le souhaitez, vous avez le TP de première année à votre disposition :
https://prepabellevue.org/index.php?article=5175

Question 7: Mettre en place le code permettant soit d'afficher que le chemin n'existe pas, soit de remonter le chemin menant à l'arrivée, en affichant dans la console les cases empruntées, la distance parcourue, et le nombre d'itérations réalisées.

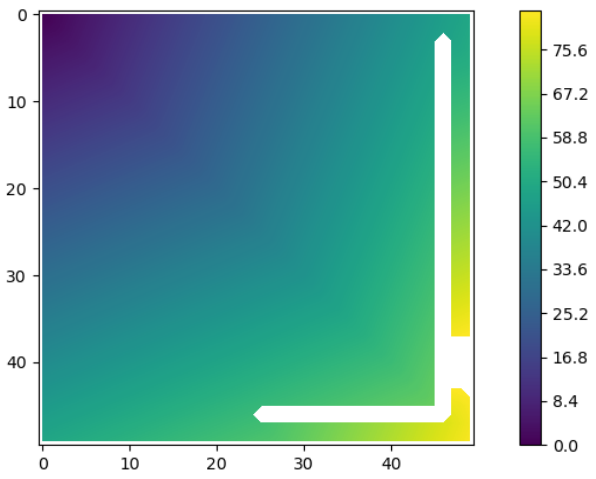
Vérifier :

```
Distance:  89.79898987322329
Itérations:  2426
['0-0', '1-0', '2-0', '3-0', '4-0', '5-0', '6-0', '7-0', '8-0', '9-0', '10-0', '11-0
', '12-0', '13-0', '14-0', '15-0', '16-0', '17-0', '18-0', '19-0', '20-0', '21-0', '
22-0', '23-0', '24-0', '25-0', '26-0', '27-0', '28-0', '29-0', '30-0', '31-0', '32-0
', '33-0', '34-0', '35-0', '36-1', '37-2', '38-3', '39-4', '40-5', '41-6', '42-7', '
43-8', '44-9', '45-10', '45-11', '45-12', '45-13', '45-14', '45-15', '45-16', '45-17
', '45-18', '45-19', '45-20', '45-21', '45-22', '45-23', '45-24', '45-25', '45-26',
'45-27', '45-28', '45-29', '45-30', '45-31', '45-32', '45-33', '45-34', '45-35', '45
-36', '45-37', '45-38', '45-39', '45-40', '45-41', '45-42', '45-43', '45-44', '45-45
', '46-46', '47-47', '48-48', '49-49']
```

Question 8: Mettre en place le code permettant d'afficher le chemin trouvé en vert sur l'image



Question 9: En utilisant la fonction Affiche_Degrade, afficher les distances des pixels depuis le départ et les pixels traités par l'algorithme



Mise en place de l'algorithme de A-star

Pour aider à la recherche du chemin le plus court, l'algorithme A* reprend l'idée de l'algorithme de Dijkstra mais ajoute l'idée introduite par l'algorithme *Best-First* détaillé ci-après. Une revue des différentes heuristiques existantes est disponible sur le site de Redblobgames [3]. On propose d'en étudier 4 dans cette section.

Afin de trouver un compromis entre rapidité et précision permettant au jeu vidéo d'être plus rapide, une bonne heuristique doit être choisie. L'expérience montre qu'il n'y a pas d'heuristique parfaite et que celle-ci dépend de la topologie de la carte considérée. Toutefois dans le cas où il n'y a aucun obstacle sur la carte, le chemin le plus court est alors obtenu en ligne droite. Ainsi sur une grille sans obstacle on utilisera :

- une heuristique Manhattan si les déplacements autorisés sont horizontaux ou verticaux ;
- une heuristique diagonale comme l'heuristique euclidienne, octile ou Chebyshev si les déplacements diagonaux sont aussi autorisés.

Ce sont simplement différentes manières de calculer une distance entre 2 points de coordonnées connues.

L'heuristique Manhattan

Le nom s'inspire des taxis newyorkais qui doivent longer les bâtiments rectangulaires pour arriver à destination, mathématiquement cette heuristique se présente simplement sous la forme suivante :

$$\forall M_i(x_i, y_i), \quad d_{Man}(M_j, M_k) = |x_j - x_k| + |y_j - y_k| \tag{1}$$

Les heuristiques diagonales

L'heuristique euclidienne

$$\forall M_i(x_i, y_i), \quad d_{euc}(M_j, M_k) = \sqrt{(x_j - x_k)^2 + (y_j - y_k)^2} \tag{2}$$

L'heuristique octile

$$\forall M_i(x_i, y_i), \quad d_{oct}(M_j, M_k) = \sqrt{2} \cdot \min\left(|x_j - x_k|, |y_j - y_k|\right) + |x_j - x_k| - |y_j - y_k| \tag{3}$$

L'heuristique Chebyshev

$$\forall M_i(x_i, y_i), \quad d_{oct}(M_j, M_k) = \max\left(|x_j - x_k|, |y_j - y_k|\right) \tag{4}$$

Question 10 : Ecrire une fonction `heuristique(pt1,pt2,type)` qui prend en entrées les coordonnées des points `pt1` et `pt2` ainsi que la variable `type` qui prend la valeur :

- 1 si on veut une heuristique Manhattan ;
- 2 si on veut une heuristique euclidienne ;
- 3 si on veut une heuristique octile ;
- 4 si on veut une heuristique Chebyshev ;

Vous travaillerez dans le fichier nommé « 11-3 - 5 - A star - Elèves.py ».

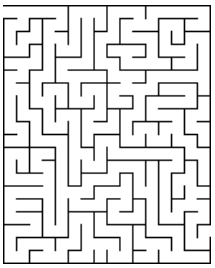
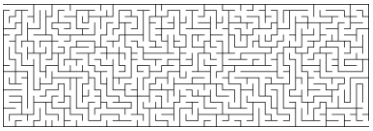
Dijkstra choisissait s comme la station la plus proche du départ. On propose de prendre en compte l'heuristique entre s et l'arrivée.

$$f(s) = d(s) + heuristique\left(s, s_{fin}, type\right)$$

Question 10: Créer la fonction f(Case) renvoyant le calcul attendu
Question 11: Copier/coller le code Dijkstra et modifier ce qu'il faut afin de réaliser l'algorithme A star
Question 12: Comparer les algorithmes Dijkstra et A star

Pour aller plus loin

Vous avez à disposition trois images :

Labyrinthe petit	Labyrinthe grand	Réseau
		

Vous trouverez dans le dossier les 3 images ci-dessus, partagées sous les formats BMP (ouverture avec matplotlib) et array (ouverture avec numpy en cas de bug matplotlib).
Pour chacun de ces 3 cas :

Question 13: Créer un nouveau fichier « 11-2 – 2 – Nom_à_choisir.py »
Question 14: Sous Python, ouvrir l'image et l'afficher
Question 15: Créer une fonction f_NB(im) qui renvoie une nouvelle image en noir et blanc ([0,0,0] ou [255,255,255]) à partir de l'image im
Attention : les labyrinthes seront aussi transformés en noir et blanc (vérifier que les triplets sont bien composés uniquement de 0 ou de 255).
Question 16: Transformer l'image en noir et blanc et l'afficher
Question 17: Choisir et ajouter les points de départ et d'arrivée sur l'image
Question 18: Faire tourner les algorithmes Dijkstra et A-star
Question 19: Apprécier les résultats et comparer l'efficacité des algorithmes

Annexes

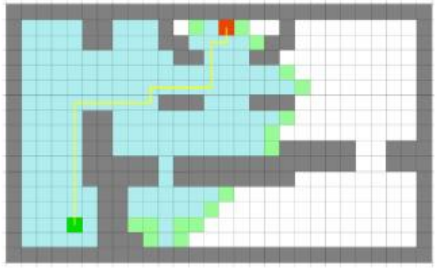
Différentes comparaisons des Algorithmes de recherches de chemin le plus court : https://qiao.github.io/PathFinding.js/visual/

	Best First Search	A*
Manhattan	Distance : 29 – Opérations : 100	Distance : 23 – Opérations : 142
Euclidienne	Distance : 23 – Opérations : 79	Distance : 23 – Opérations : 201
Octile	Distance : 23 – Opérations : 79	Distance : 23 – Opérations : 194
Chebyshev	Distance : 23 – Opérations : 79	Distance : 23 – Opérations : 223
Dijkstra	Distance : 23 – Opérations : 348	

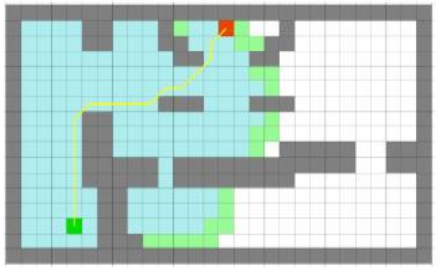
TABLEAU ANNEXE 2 – Distances calculées (en unités) et opérations effectuées lorsque les diagonales sont **interdites**

	Best First Search	A*
Manhattan	Distance : 22,56 – Opérations : 88	Distance : 20,07 – Opérations : 100
Euclidienne	Distance : 22,56 – Opérations : 85	Distance : 20,07 – Opérations : 160
Octile	Distance : 22,56 – Opérations : 85	Distance : 20,07 – Opérations : 147
Chebyshev	Distance : 20,73 – Opérations : 77	Distance : 20,07 – Opérations : 192
Dijkstra	Distance : 20,07 – Opérations : 346	

TABLEAU ANNEXE 3 – Distances calculées (en unités) et opérations effectuées lorsque les diagonales sont **autorisées**

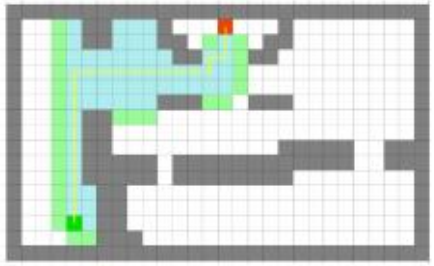


(a) Diagonales interdites
distance calculée : 23 – opérations effectuées : 348

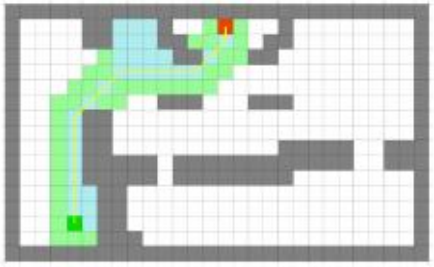


(b) Diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 346

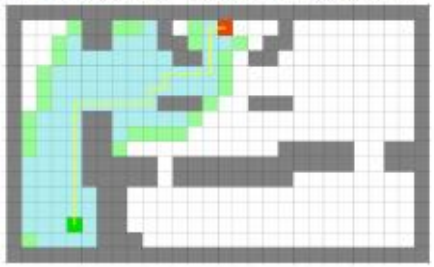
1ER NIVEAU ANNEXE 1 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant l'algorithme de Dijkstra



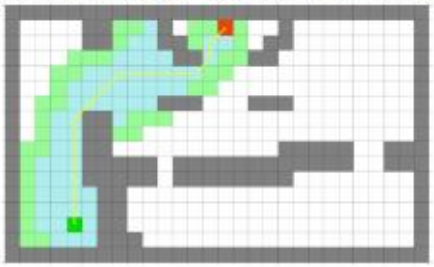
(a) Heuristique Manhattan – Diagonales interdites
distance calculée : 23 – opérations effectuées : 142



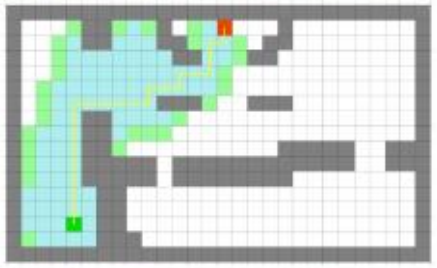
(b) Heuristique Manhattan – Diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 100



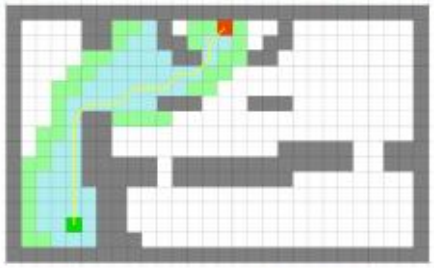
(c) Heuristique euclidienne – diagonales interdites
distance calculée : 23 – opérations effectuées : 201



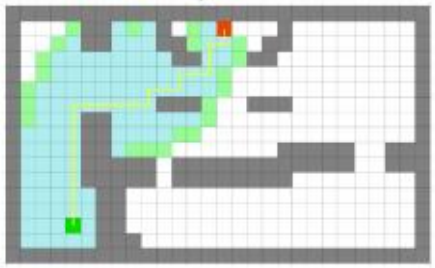
(d) Heuristique euclidienne – diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 160



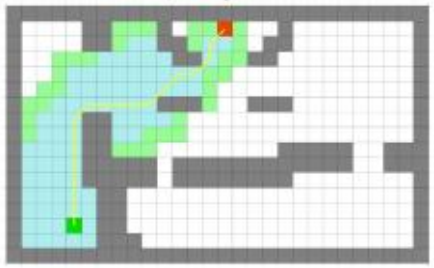
(e) Heuristique octile – diagonales interdites
distance calculée : 23 – opérations effectuées : 194



(f) Heuristique octile – diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 147

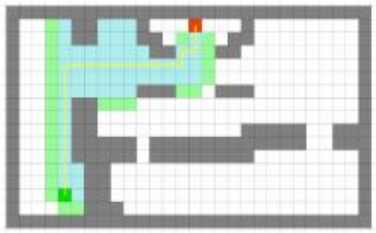


(g) Heuristique Chebyshev – diagonales interdites
distance calculée : 23 – opérations effectuées : 223

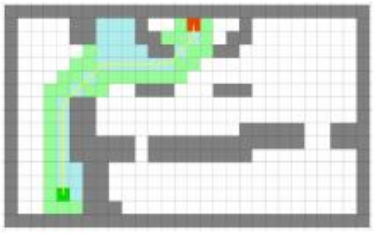


(h) Heuristique Chebyshev – diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 192

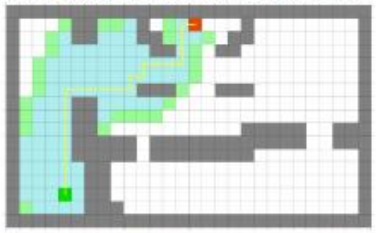
1ER NIVEAU ANNEXE 2 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant une stratégie *Best First Search*



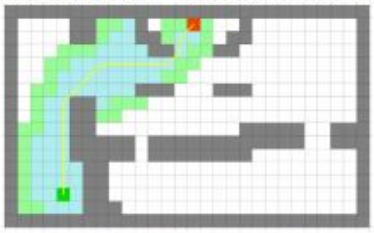
(a) Heuristique Manhattan – Diagonales interdites
distance calculée : 23 – opérations effectuées : 142



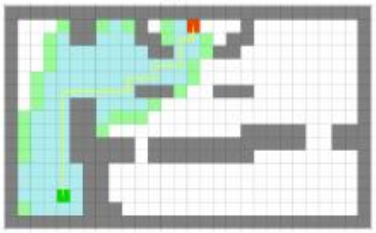
(b) Heuristique Manhattan – Diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 100



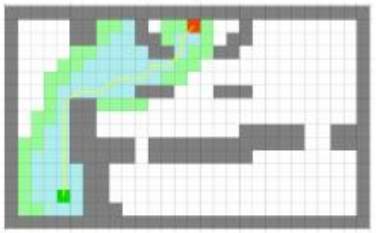
(c) Heuristique euclidienne – diagonales interdites
distance calculée : 23 – opérations effectuées : 201



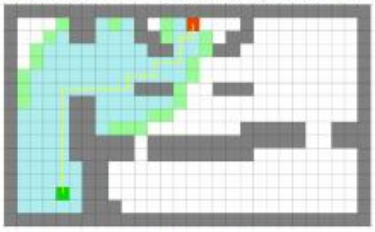
(d) Heuristique euclidienne – diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 160



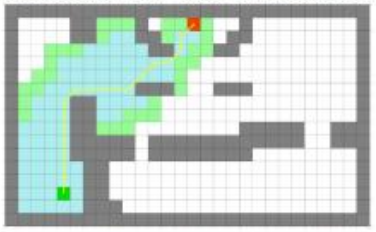
(e) Heuristique octile – diagonales interdites
distance calculée : 23 – opérations effectuées : 194



(f) Heuristique octile – diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 147



(g) Heuristique Chebyshev – diagonales interdites
distance calculée : 23 – opérations effectuées : 223



(h) Heuristique Chebyshev – diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 192

1ER NIVEAU ANNEXE 3 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant l'algorithme A* pour plusieurs heuristiques

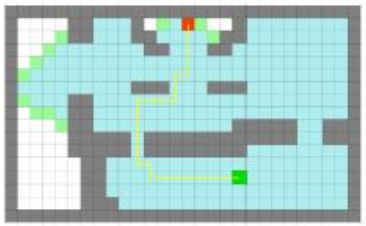
ANNEXE : DEUXIEME NIVEAU

	Best First Search	A*
Manhattan	Distance : 24 – Opérations : 101	Distance : 24 – Opérations : 182
Euclidienne	Distance : 24 – Opérations : 79	Distance : 24 – Opérations : 334
Octile	Distance : 24 – Opérations : 85	Distance : 24 – Opérations : 296
Chebyshev	Distance : 26 – Opérations : 82	Distance : 24 – Opérations : 370
Dijkstra	Distance : 24 – Opérations : 531	

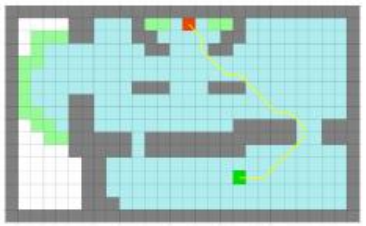
TABLEAU ANNEXE 4 – Distances calculées (en unités) et opérations effectuées lorsque les diagonales sont **interdites**

	Best First Search	A*
Manhattan	Distance : 20,49 – Opérations : 98	Distance : 20,14 – Opérations : 146
Euclidienne	Distance : 20,49 – Opérations : 78	Distance : 20,14 – Opérations : 233
Octile	Distance : 20,49 – Opérations : 82	Distance : 20,14 – Opérations : 186
Chebyshev	Distance : 20,73 – Opérations : 77	Distance : 20,14 – Opérations : 326
Dijkstra	Distance : 20,14 – Opérations : 547	

TABLEAU ANNEXE 5 – Distances calculées (en unités) et opérations effectuées lorsque les diagonales sont **autorisées**

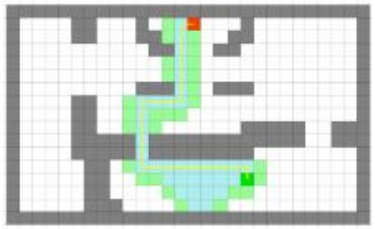


(a) Diagonales interdites
distance calculée : 24 – opérations effectuées : 531

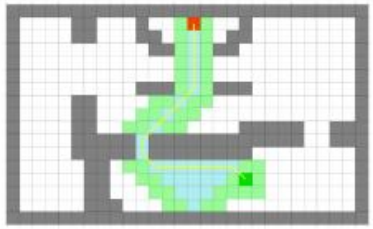


(b) Diagonales autorisées
distance calculée : 20,14 – opérations effectuées : 547

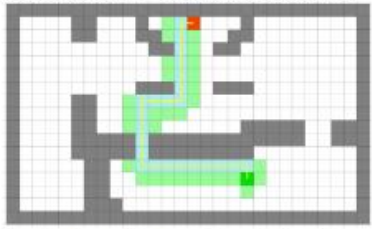
2ÈME NIVEAU ANNEXE 1 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant l'algorithme de Dijkstra



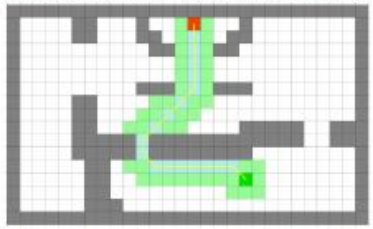
(a) Heuristique Manhattan – Diagonales interdites
distance calculée : 24 – opérations effectuées : 101



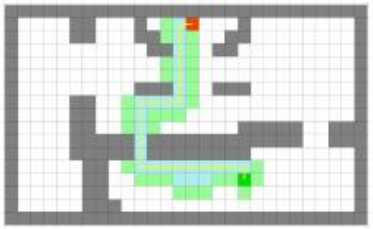
(b) Heuristique Manhattan – Diagonales autorisées
distance calculée : 20,49 – opérations effectuées : 98



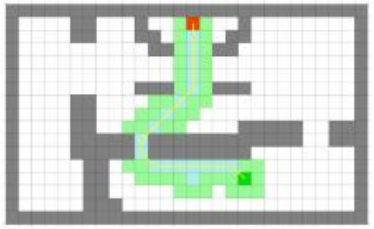
(c) Heuristique euclidienne – diagonales interdites
distance calculée : 24 – opérations effectuées : 79



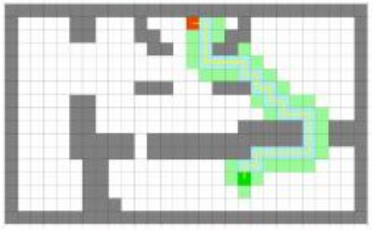
(d) Heuristique euclidienne – diagonales autorisées
distance calculée : 20,49 – opérations effectuées : 78



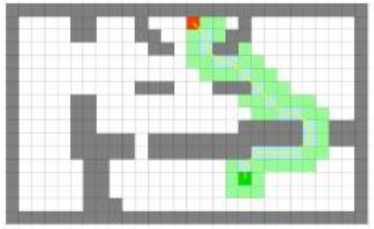
(e) Heuristique octile – diagonales interdites
distance calculée : 24 – opérations effectuées : 85



(f) Heuristique octile – diagonales autorisées
distance calculée : 20,49 – opérations effectuées : 82

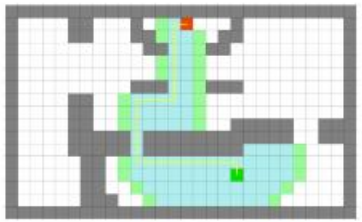


(g) Heuristique Chebyshev – diagonales interdites
distance calculée : 26 – opérations effectuées : 82

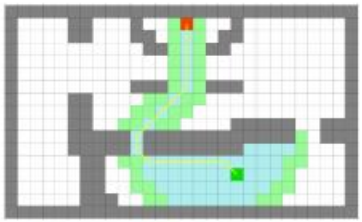


(h) Heuristique Chebyshev – diagonales autorisées
distance calculée : 20,73 – opérations effectuées : 77

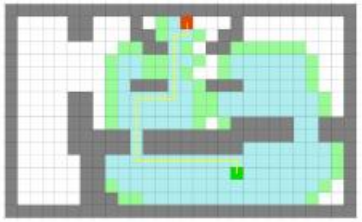
2ÈME NIVEAU ANNEXE 2 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant une stratégie *Best First Search*



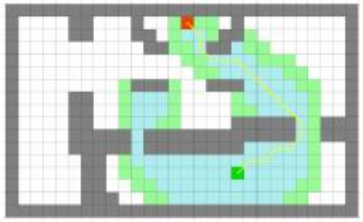
(a) Heuristique Manhattan – Diagonales interdites
distance calculée : 24 – opérations effectuées : 182



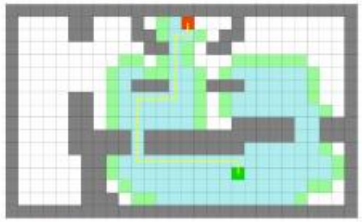
(b) Heuristique Manhattan – Diagonales autorisées
distance calculée : 20,14 – opérations effectuées : 146



(c) Heuristique euclidienne – diagonales interdites
distance calculée : 24 – opérations effectuées : 334

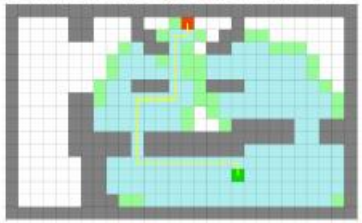


(d) Heuristique euclidienne – diagonales autorisées
distance calculée : 20,14 – opérations effectuées : 233

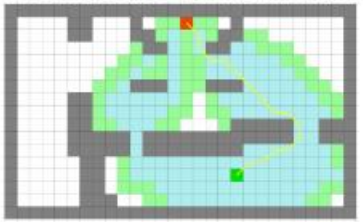


(e) Heuristique octile – diagonales interdites
distance calculée : 24 – opérations effectuées : 296

(f) Heuristique octile – diagonales autorisées
distance calculée : 20,14 – opérations effectuées : 186



(g) Heuristique Chebyshev – diagonales interdites
distance calculée : 24 – opérations effectuées : 370



(h) Heuristique Chebyshev – diagonales autorisées
distance calculée : 20,14 – opérations effectuées : 326

2ÈME NIVEAU ANNEXE 3 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant l'algorithme A* pour plusieurs heuristiques