TP6 - Recherche du plus court chemin par l'algorithme A*

I Mise en situation

I.1 Un *Tactical RPG* : *Fire Emblem Three Houses*

Parmi les différentes catégories de RPG (*Role Playing Game*) existantes, nous nous intéresserons dans ce TP au *Tactical RPG* en tour par tour. Le *gameplay* consiste alors à prendre des décisions judicieuses lors du tour du joueur pour remporter des combats de grande envergure sur une carte qui sert de grille de jeu. Une victoire rapporte des points d'expérience qui permettent aux troupes victorieuses de devenir plus puissantes.

Nous prendrons pour support dans ce TP le jeu *Fire Emblem Three Houses*, (1) et nous nous intéresserons plus particulièrement à l'Intelligence Artificielle implémentée pour gérer les mouvements des personnages non jouables.

FIGURE 1 – Visuel du jeu *Fire Emblem Three Houses* sorti le 26 Juillet 2019

I.2 Un système de grille pour le déplacement des personnages

Lors de la sélection d'une troupe sur la carte (en surbrillance dorée sur la 2), un quadrillage apparaît. Les cases bleues correspondent aux déplacements possibles de la troupe, les cases rouges indiquent la portée de l'attaque si la troupe se déplace sur les cases bleues les plus éloignées. Une troupe constituée d'archers aura évidemment plus de portée qu'une troupe d'épéistes.

FIGURE 2 – Possibilité de déplacement d'une troupe sélectionnée.

On peut observer qu'il n'est pas possible de passer par les bâtiments et que les forêts réduisent la distance que la troupe peut parcourir lors de son tour.

I.3 Problématique de l'Intelligence Artificielle des troupes ennemies et du confort de jeu

Dans un soucis de dynamisme dans le jeu et dans sa gestion de la difficulté, les développeurs souhaitent implémenter une Intelligence Artificielle pour les troupes adverses capable de trouver le plus court chemin pour attaquer une troupe alliée. En effet si le calcul de l'itinéraire est trop long, le jeu

est ralenti et l'expérience joueur s'en trouve dégradée. On pose donc la problématique suivante :

Problématique technique

Quel algorithme du plus court chemin privilégier pour limiter le temps de calcul de l'itinéraire d'une troupe gérée par la console ?

La gestion de la difficulté est plutôt simple : il est possible d'ajouter de l'erreur dans l'algorithme pour que la troupe ne choisisse pas forcément le meilleur trajet. La difficulté pour les développeurs est de suffisamment bien cacher cette "bêtise" pour que le joueur ne la perçoive pas.

II Un algorithme souvent employé dans le domaine du jeu vidéo : l'algorithme A*

II.1 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**).

II.2 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.

1. https://www.researchgate.net/publication/267809499_A-based_Pathfinding_in_Modern_Computer_Games
2. http://www.airccse.org/journal/ijaia/papers/1011ijaia01.pdf

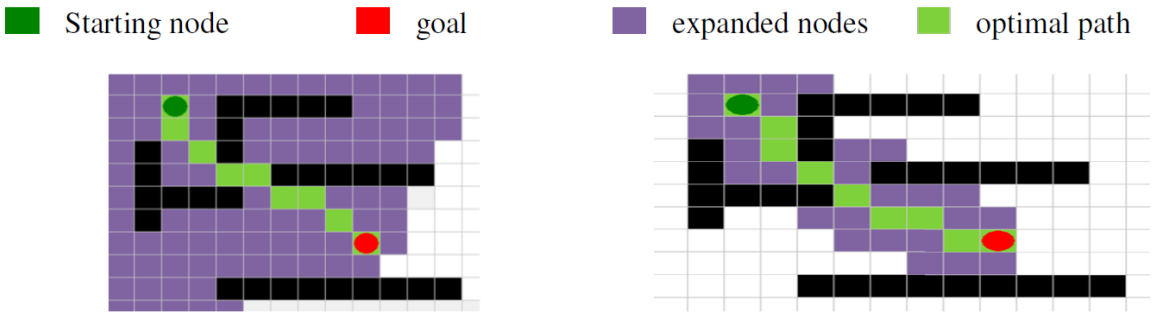


FIGURE 3 – Comparaison des algorithmes de Dijkstra et A*. L'algorithme de Dijkstra envisage beaucoup plus de points à visiter.

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.

III Carte considérée pour ce TP

On se base sur la carte donnée en 4, Celle-ci est quadrillée. La troupe alliée est indiquée en **vert** (coordonnée **L14C4**) et la troupe adverse est indiquée en **rouge** (coordonnée **L1C14**). Les cases accessibles par les troupes sont numérotées **1** et les cases non accessibles (à cause d'un mur ou d'un cours d'eau) sont indiquées par un **0**. Le quadrillage est aussi disponible sous forme d'un fichier texte nommé `Matrice_carte.prn`.

	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	

FIGURE 4 – Quadrillage de la carte du niveau considéré.

Objectif

On cherche dans un premier temps à traduire ce quadrillage en matrice d'adjacence en vue d'obtenir le graphe correspondant sous `Graphviz`.

Dans cette partie du TP on utilisera la trame fournie dans le code `Partie3.py` du dossier `Codes\Enonce\Partie_3`.

III.1 Echauffement : matrice d'adjacence pour un quadrillage simple

On considère dans un premier temps le quadrillage de la 5 de fichier texte associé `Test.prn`.

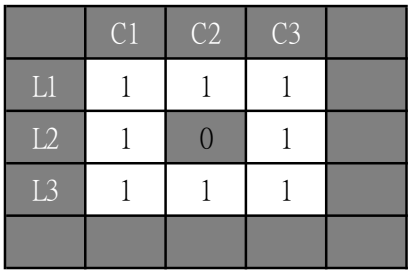


FIGURE 5 – Quadrillage test en vue d'écrire la matrice d'adjacence.

On cherche à obtenir la matrice d'adjacence associée à ce quadrillage dans 2 cas :

- lorsqu'on autorise uniquement les déplacements horizontaux et verticaux d'une case à l'autre ;
- lorsqu'on autorise en plus les déplacements diagonaux.

III.1.1 Matrice d'adjacence obtenue pour des déplacements horizontaux ou verticaux uniquement

On cherche à écrire la matrice d'adjacence sous la forme suivante :

	L1C1	L1C2	L1C3	L2C1	L2C2	L2C3	L3C1	L3C2	L3C3
L1C1	0	1	0	1	0	0	0	0	0
L1C2	...	...	...	...	...	...	...	...	...
L1C3	...	...	...	...	...	...	...	...	...
L2C1	...	...	...	...	...	...	...	...	...
L2C2	...	...	...	...	...	...	...	...	...
L2C3	...	...	...	...	...	...	...	...	...
L3C1	...	...	...	...	...	...	...	...	...
L3C2	...	...	...	...	...	...	...	...	...
L3C3	...	...	...	...	...	...	...	...	...

TABLE 1 – Forme désirée de la matrice d'adjacence

La case de coordonnées (Li,Ci) est notée LiCi dans la matrice. Par exemple la case de coordonnées (L1,C1) notée L1C1 dans le `Forme_mat_adj` possède pour voisins la case L1C2 ainsi que la case L2C1. On n'autorisera pas les déplacements diagonaux ici. Enfin on supposera que la distance entre 2 cases adjacentes par déplacement horizontal ou vertical sera de 1.

Q1. Compléter sur le Document Réponses `DR1` la matrice d'adjacence associée au quadrillage de la 5 d'après la trame expliquée précédemment.
Corrige

	L1C1	L1C2	L1C3	L2C1	L2C2	L2C3	L3C1	L3C2	L3C3
L1C1	0	1	0	1	0	0	0	0	0
L1C2	1	0	1	0	0	0	0	0	0
L1C3	0	1	0	0	0	1	0	0	0
L2C1	1	0	0	0	0	0	1	0	0
L2C2	0	0	0	0	0	0	0	0	0
L2C3	0	0	1	0	0	0	0	0	1
L3C1	0	0	0	1	0	0	0	1	0
L3C2	1	0	1	0	0	0	1	0	1
L3C3	1	0	1	0	0	1	0	1	0

Q2. Utiliser la fonction `mat_adj(fichier_texte)` proposée dans le script `Partie3.py` pour lire le fichier texte `Test.prn` et vérifier qu'on obtient bien la même matrice.
Corrige

```
test = mat_adj("Test.prn")
print(test)
>>> [[0. 1. 0. 1. 0. 0. 0. 0. 0.]
          [1. 0. 1. 0. 0. 0. 0. 0. 0.]
          [0. 1. 0. 0. 0. 1. 0. 0. 0.]
          [1. 0. 0. 0. 0. 0. 1. 0. 0.]
          [0. 0. 0. 0. 0. 0. 0. 0. 0.]
          [0. 0. 1. 0. 0. 0. 0. 0. 1.]
          [0. 0. 0. 1. 0. 0. 0. 1. 0.]
          [0. 0. 0. 0. 0. 0. 1. 0. 1.]
          [0. 0. 0. 0. 0. 1. 0. 1. 0.]]
```

III.1.2 Matrice d'adjacence obtenue pour des déplacements diagonaux autorisés

On cherche cette fois-ci à obtenir la matrice d'adjacence lorsque les déplacements diagonaux sont autorisés. La distance à parcourir pour se déplacer d'une case en diagonale est de $\sqrt{2}$.

Si on regarde la 5 on remarque les 2 types de diagonales suivantes :

- on peut passer de la case L2C1 à la case L1C2 (inversion des numéros), on a alors une diagonale $\nearrow$;
- on peut passer de la case L1C2 à la case L2C3 (incrément d'une ligne et d'une colonne), on a alors une diagonale $\searrow$.

Q3. Ecrire une fonction `mat_adj_diag` qui reprend la trame de la fonction `mat_adj` mais qui prend en plus en compte la possibilité de se déplacer en diagonale.
Corrige

```python
def mat_adj(fichier_texte):
    # Lecture du fichier texte
    with open(fichier_texte, 'r') as file :
        text=file.readlines()
    # Nombre de lignes dans le fichier texte, on retire 1 pour enlever la
    # première ligne qui ne contient que le nom des colonnes
    nb_lignes = len(text)-1
    # Création d'une liste contenant le nom des colonnes
    nom_colonne = text[0].split()
    nb_col = len(nom_colonne)
    # Taille de la matrice à écrire et définition de la matrice
    taille_matrice = nb_lignes*nb_col
    mat_adj = np.zeros((taille_matrice,taille_matrice))
    #-----------------------------------------
    # Recherche des cases adjacentes
    for l in range(nb_lignes-1):
        # On relève la ligne actuelle et on supprime son premier terme
        ligne_actuelle = text[l+1].split()
        ligne_actuelle = ligne_actuelle[1:]
        # On relève la ligne suivante et on supprime son premier terme
        ligne_suivante = text[l+2].split()
        ligne_suivante = ligne_suivante[1:]

        for c in range(nb_col-1):
            #-----------------------------------------
            # Traitement de la ligne actuelle, on regarde à droite si la valeur
            # vaut 1, si c'est le cas la case est accessible
            if int(ligne_actuelle[c+1]) == 1 and int(ligne_actuelle[c]) != 0:
                mat_adj[nb_col*l+c][nb_col*l+c+1] =1
                # Symétrie de la matrice d'adjacence
                mat_adj[nb_col*l+c+1][nb_col*l+c] =1

            # On termine avec la dernière ligne
            if l == nb_lignes-2:
                if int(ligne_suivante[c+1]) == 1 and int(ligne_suivante[c]) != 0:
                    mat_adj[nb_col*(l+1)+c][nb_col*(l+1)+c+1] =1
                    # Symétrie de la matrice d'adjacence
                    mat_adj[nb_col*(l+1)+c+1][nb_col*(l+1)+c] =1

            #-------------------------------------
            # Traitement de la ligne suivante, on regarde si on a un 1 sur la
            # ligne suivante pour un même numéro de colonne
            if int(ligne_suivante[c]) == 1 and int(ligne_actuelle[c]) != 0:
                mat_adj[nb_col*l+c][nb_col*(l+1)+c] =1
                # Symétrie de la matrice d'adjacence
                mat_adj[nb_col*(l+1)+c][nb_col*l+c] =1

            # On termine avec la dernière colonne
            if c ==nb_col-2 and int(ligne_suivante[nb_col-1]) == 1 \
            and int(ligne_actuelle[nb_col-1]) != 0:
                mat_adj[nb_col*l+nb_col-1][nb_col*(l+1)+nb_col-1] =1
                # Symétrie de la matrice d'adjacence
                mat_adj[nb_col*(l+1)+nb_col-1][nb_col*l+nb_col-1] =1

    return mat_adj
```

IV Différentes heuristiques au choix pour notre problème

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.

IV.1 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 :

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

IV.2 les heuristiques diagonales

IV.2.1 L'heuristique euclidienne

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

IV.2.2 L'heuristique octile

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

IV.2.3 L'heuristique Chebyshev

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

IV.3 Ecriture d'une fonction `heuristique`

Les cartes étudiées étant présentées sous forme de quadrillage, le système de coordonnées est très simple :

- le point de coordonnées L1C1 aura pour coordonnées cartésiennes (0,0) ;
- le point de coordonnées L1C2 aura pour coordonnées cartésiennes (1,0) ;

3. `http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html`

- le point de coordonnées L2C1 aura pour coordonnées cartésiennes (0,1).

Q4. 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 ;

Corrige

```
# Mise en forme des coordonnées des points
pt1 = [X1,Y1]
pt2 = [X2,Y2]

def heuristique(pt1,pt2,type):
        if type == 1:
                return abs(pt1[0] - pt2[0]) + abs(pt1[1] - pt2[1])
        if type == 2:
                return sqrt((pt1[0] - pt2[0])**2 + (pt1[1] - pt2[1])**2)
        if type == 3:
                return sqrt(2)*min(abs(pt1[0] - pt2[0]),abs(pt1[1] - pt2[1])) /
                + abs(abs(pt1[0] - pt2[0]) - abs(pt1[1] - pt2[1]))
        if type == 4:
                return max(abs(pt1[0] - pt2[0]),abs(pt1[1] - pt2[1]))
```

V Ouverture

V.1 Un simulateur de recherche de plus court chemin dans un labyrinthe

Un très bon simulateur permettant de comparer différents algorithmes de plus courts chemins utilisant plusieurs heuristiques différentes est disponible au lien suivant :
`https://qiao.github.io/PathFinding.js/visual/`
Attention toutefois le simulateur proposé n'est pas bien adapté aux petits écrans.

V.2 Et pour une grille composée de cases hexagonales ?

D'autres jeux de stratégie comme *Civilization* utilisent des grilles à cases hexagonales. Le système de coordonnées adopté est alors 3D :

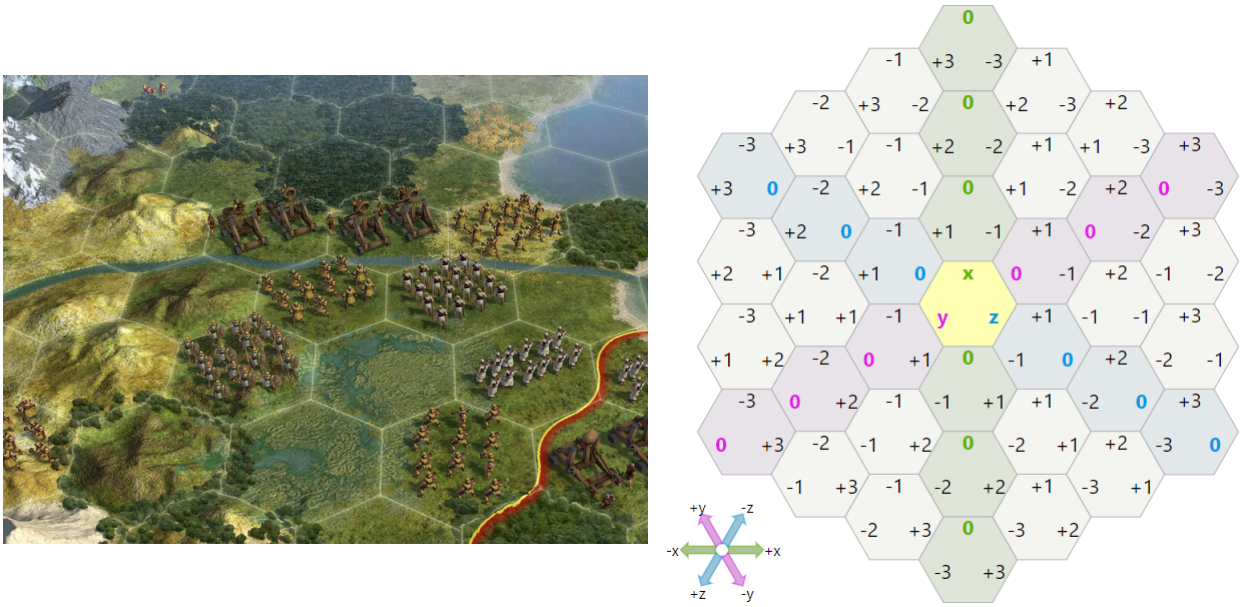


FIGURE 6 – Quadrillage du jeu *Civilization* et système de coordonnées associé

L'algorithme A* marchera de la même manière mais on préfèrera une heuristique de type "distance de Manhattan" adaptée :

```
depart = [departX,departY,departZ] # Coordonnées cartésiennes du point de départ
arrivee = [arriveeX,arriveeY,arriveeZ] # Coordonnées cartésiennes de l'arrivée

def Manhattan_hexa(depart,arrivee):
        return 1/2*(abs(arrivee[0] - depart[0]) + abs(arrivee[1] - depart[1]) /
        + abs(arrivee[2] - depart[2]))
```

Plus d'informations sont disponibles sur le site `https://www.redblobgames.com/grids/hexagons/#distances`

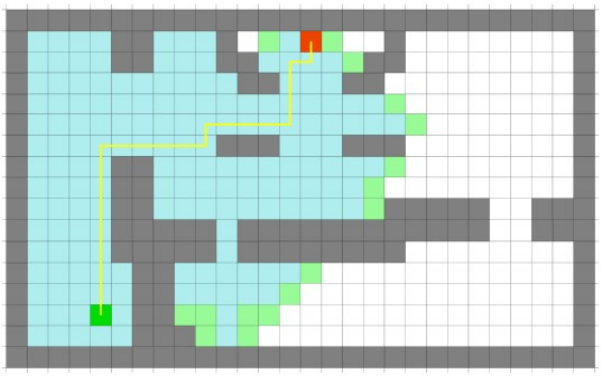
ANNEXE : PREMIER NIVEAU

	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	

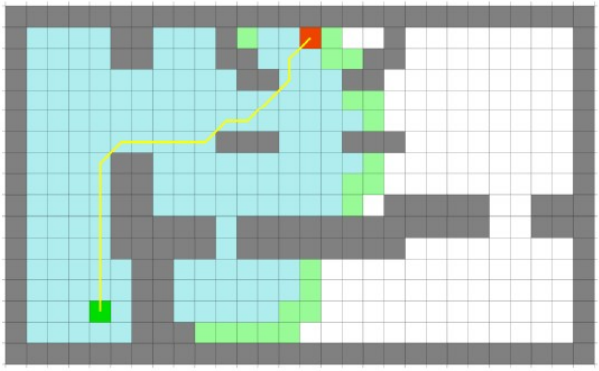
TABLEAU ANNEXE 2 – Distances calculées (en unités) et opérations effectuées lorsque les diagonales sont **interdites**

	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**

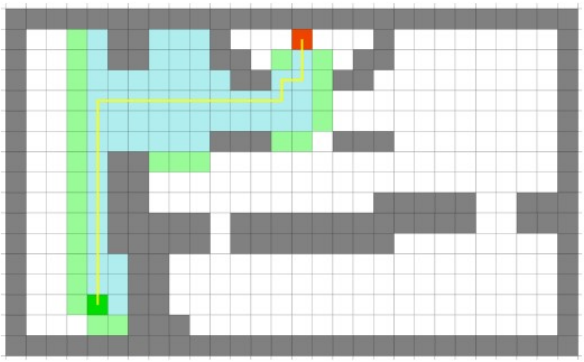


(a) Diagonales interdites
distance calculée : 23 – opérations effectuées : 348

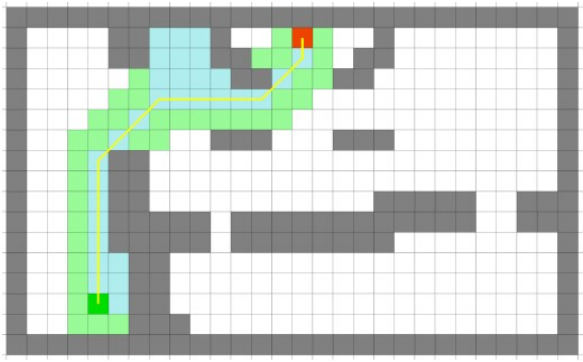


(b) Diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 346

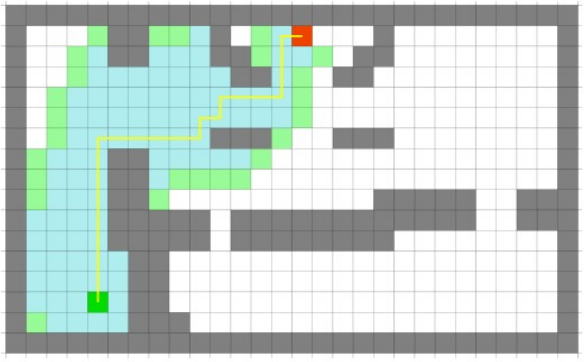
1ER NIVEAU ANNEXE 1 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant l'algorithme de Dijkstra



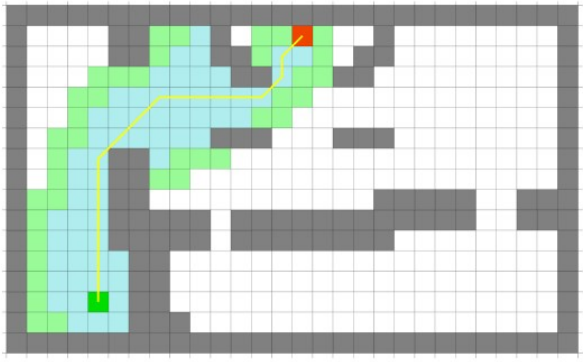
(a) Heuristique Manhattan – Diagonales interdites
distance calculée : 23 – opérations effectuées : 142



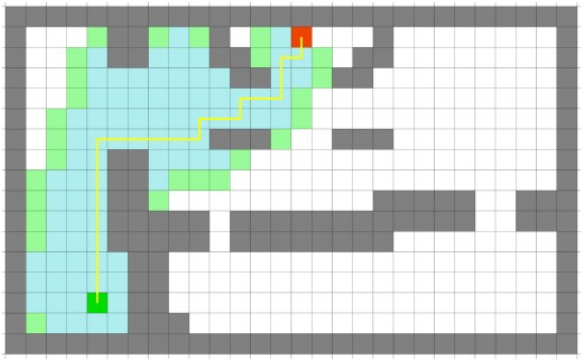
(b) Heuristique Manhattan – Diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 100



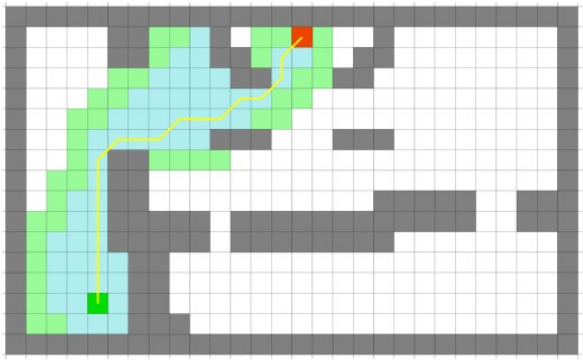
(c) Heuristique euclidienne – diagonales interdites
distance calculée : 23 – opérations effectuées : 201



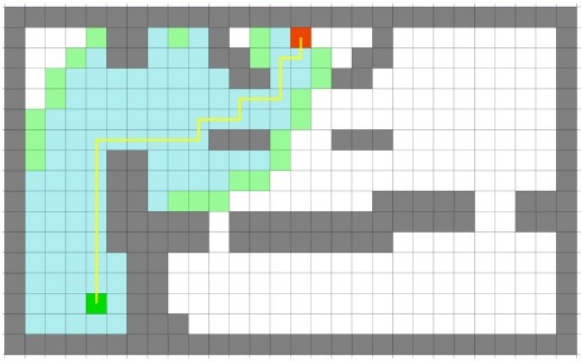
(d) Heuristique euclidienne – diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 160



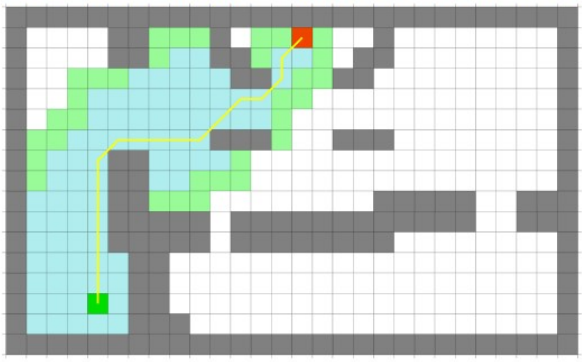
(e) Heuristique octile – diagonales interdites
distance calculée : 23 – opérations effectuées : 194



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

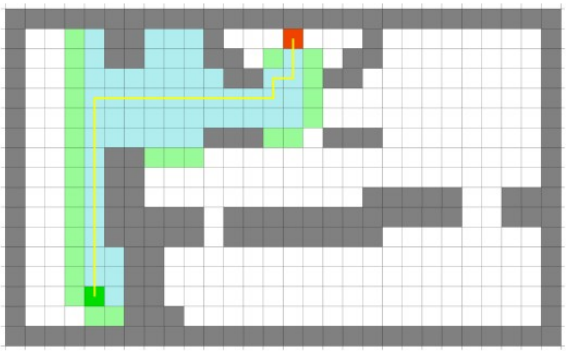


(g) Heuristique Chebyshev – diagonales interdites
distance calculée : 23 – opérations effectuées : 223

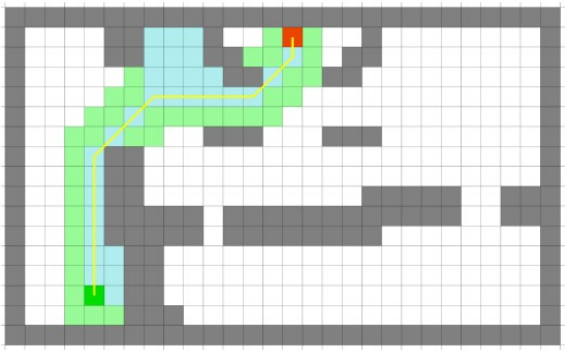


(h) Heuristique Chebyshev – diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 192

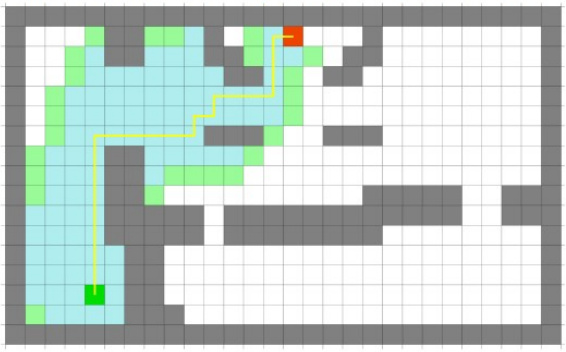
1ER NIVEAU ANNEXE 2 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant une stratégie *Best First Search*



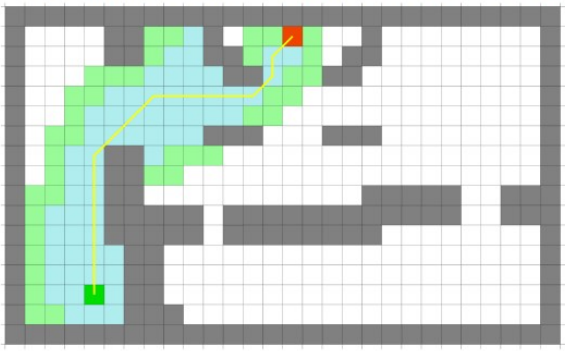
(a) Heuristique Manhattan – Diagonales interdites
distance calculée : 23 – opérations effectuées : 142



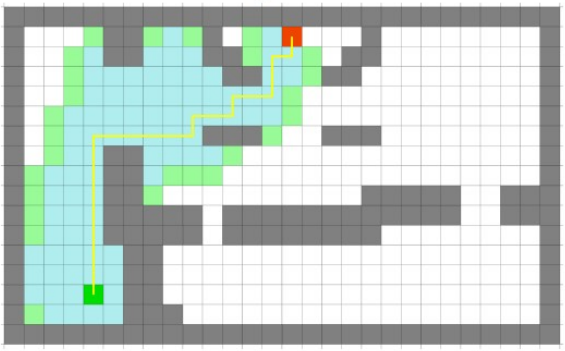
(b) Heuristique Manhattan – Diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 100



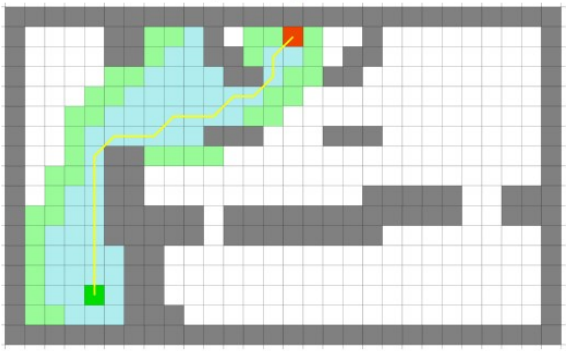
(c) Heuristique euclidienne – diagonales interdites
distance calculée : 23 – opérations effectuées : 201



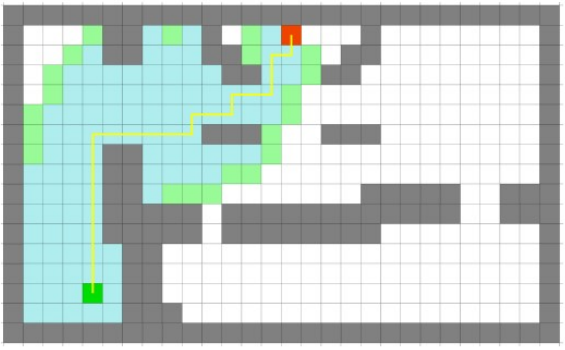
(d) Heuristique euclidienne – diagonales autorisées
distance calculée : 20,07 – opérations effectuées : 160



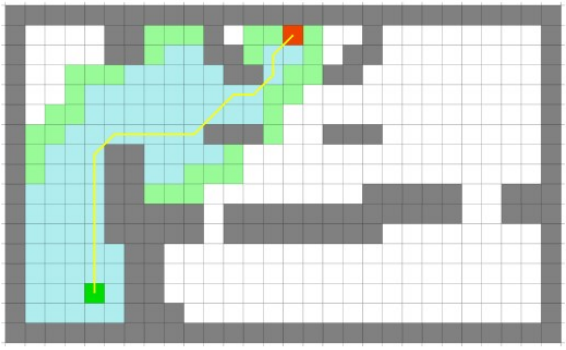
(e) Heuristique octile – diagonales interdites
distance calculée : 23 – opérations effectuées : 194



(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

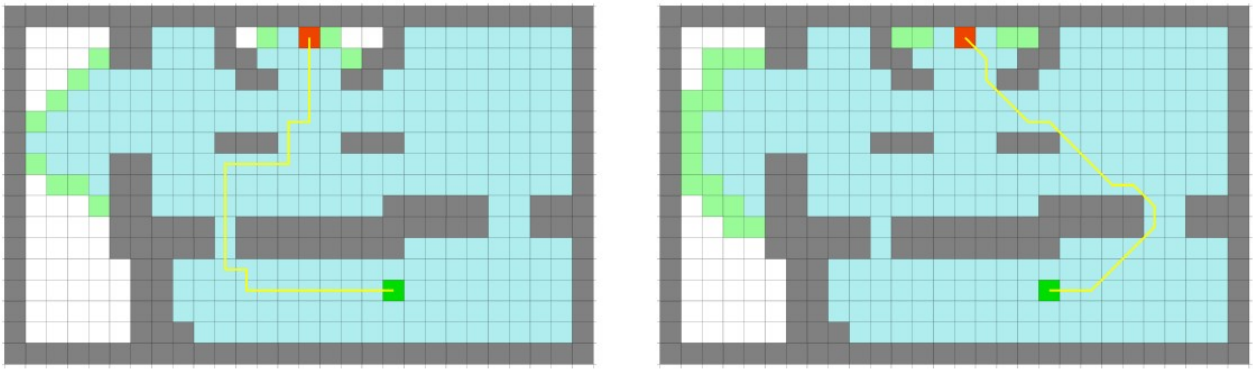
ANNEXE : DEUXIEME NIVEAU

	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	

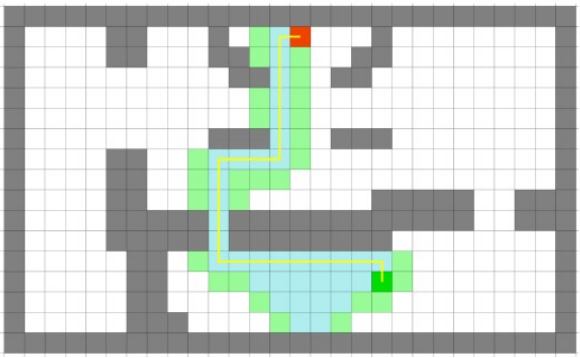
TABLEAU ANNEXE 5 – Distances calculées (en unités) et opérations effectuées lorsque les diagonales sont **autorisées**



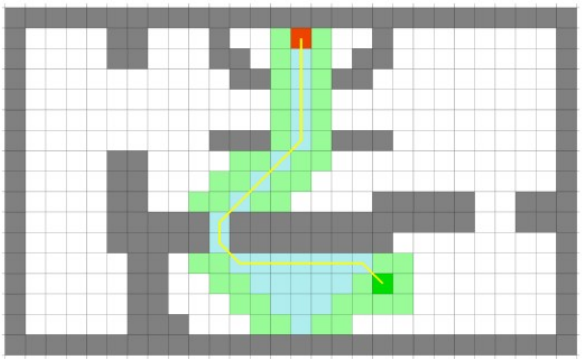
(a) Diagonales interdites
distance calculée : 24 – opérations effectuées : 531

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

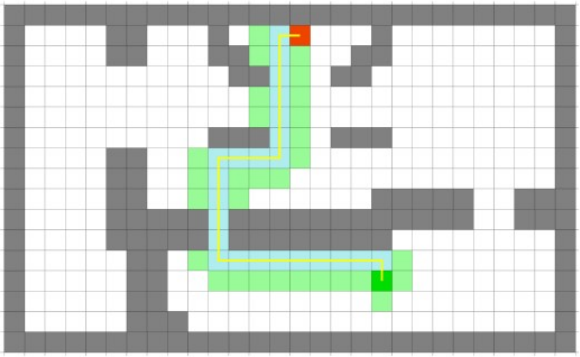
2ÈME NIVEAU ANNEXE 1 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant l'algorithme de Dijkstra



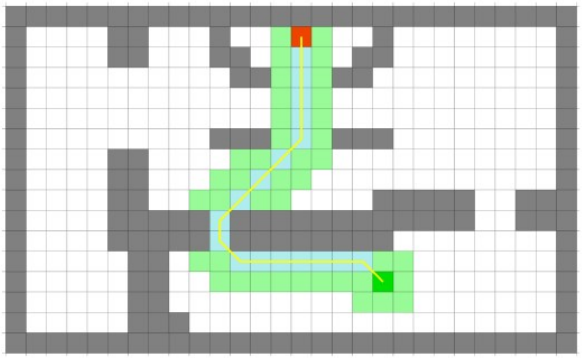
(a) Heuristique Manhattan – Diagonales interdites
distance calculée : 24 – opérations effectuées : 101



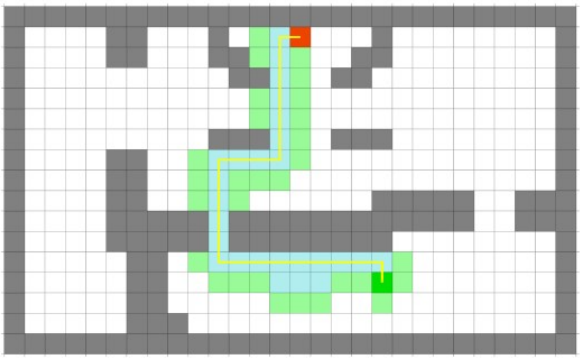
(b) Heuristique Manhattan – Diagonales autorisées
distance calculée : 20,49 – opérations effectuées : 98



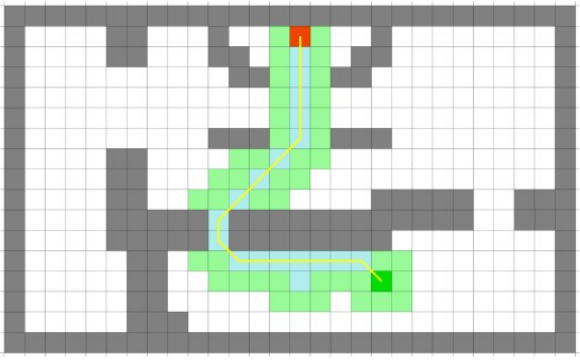
(c) Heuristique euclidienne – diagonales interdites
distance calculée : 24 – opérations effectuées : 79



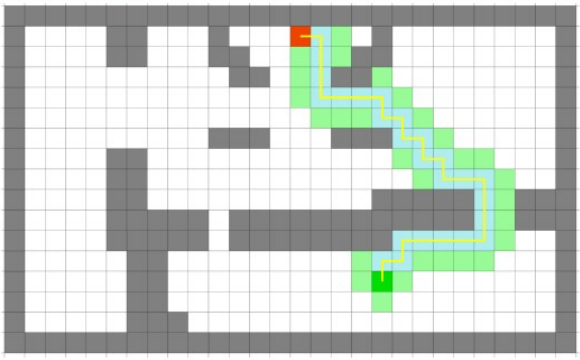
(d) Heuristique euclidienne – diagonales autorisées
distance calculée : 20,49 – opérations effectuées : 78



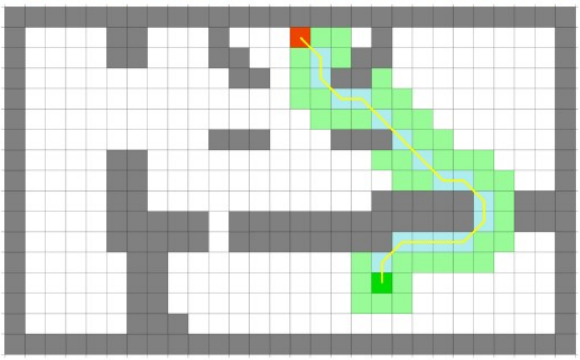
(e) Heuristique octile – diagonales interdites
distance calculée : 24 – opérations effectuées : 85



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

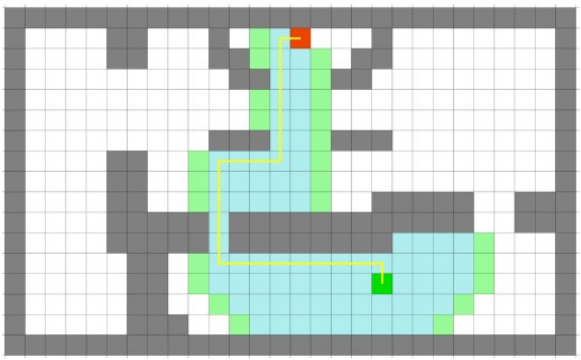


(g) Heuristique Chebyshev – diagonales interdites
distance calculée : 26 – opérations effectuées : 82

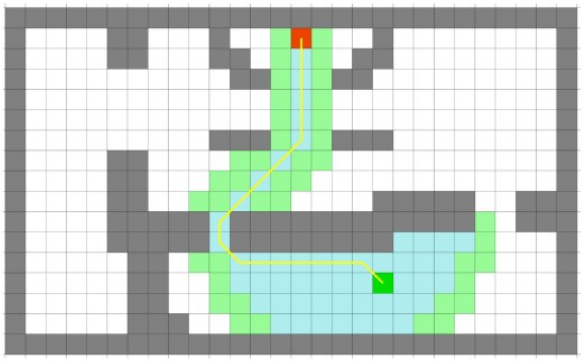


(h) Heuristique Chebyshev – diagonales autorisées
distance calculée : 20,73 – opérations effectuées : 77

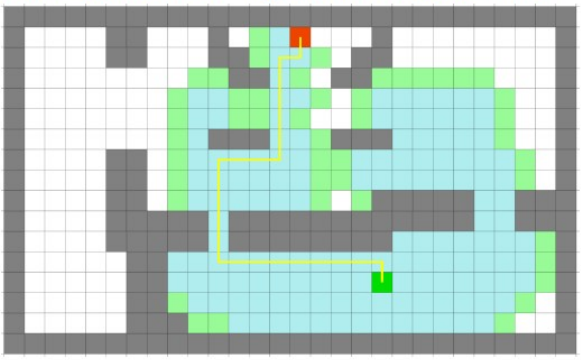
2ème niveau annexe 2 – Trajectoire trouvée par l'IA de la troupe adverse en utilisant une stratégie *Best First Search*



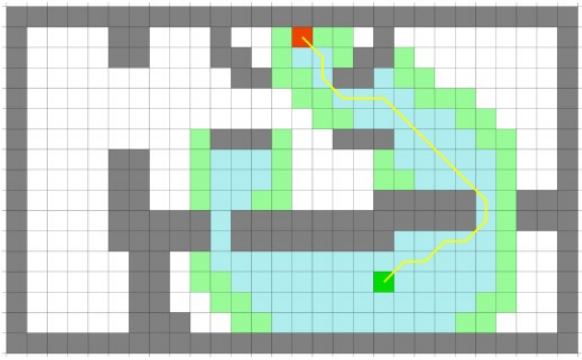
(a) Heuristique Manhattan – Diagonales interdites
distance calculée : 24 – opérations effectuées : 182



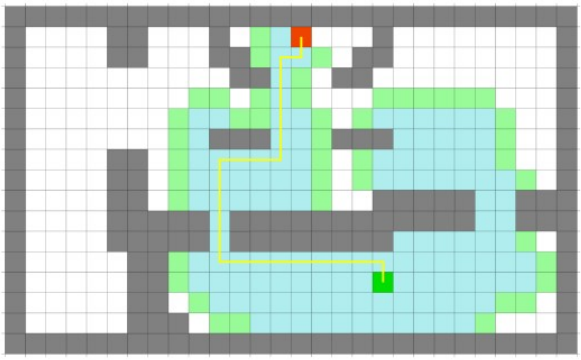
(b) Heuristique Manhattan – Diagonales autorisées
distance calculée : 20,14 – opérations effectuées : 146



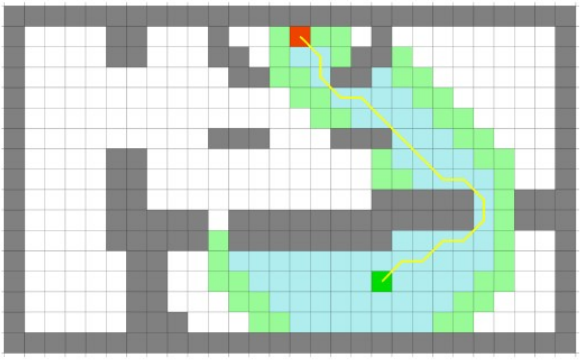
(c) Heuristique euclidienne – diagonales interdites
distance calculée : 24 – opérations effectuées : 334



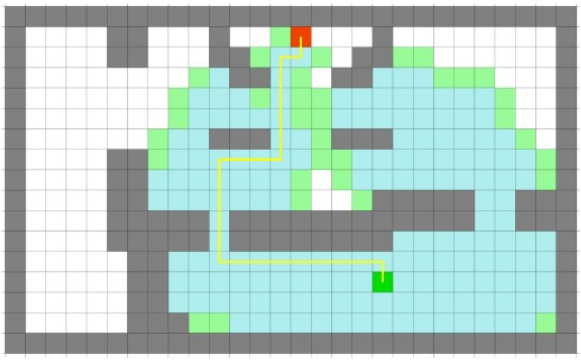
(d) Heuristique euclidienne – diagonales autorisées
distance calculée : 20,14 – opérations effectuées : 233



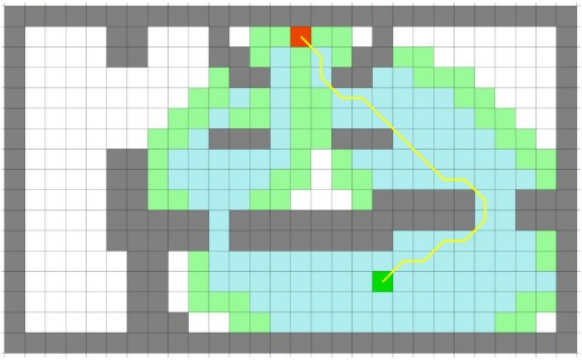
(e) Heuristique octile – diagonales interdites
distance calculée : 24 – opérations effectuées : 296



(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

Documents Réponses

Q1.

	L1C1	L1C2	L1C3	L2C1	L2C2	L2C3	L3C1	L3C2	L3C3
L1C1	0	1	0	1	0	0	0	0	0
L1C2	...	...	...	...	...	...	...	...	...
L1C3	...	...	...	...	...	...	...	...	...
L2C1	...	...	...	...	...	...	...	...	...
L2C2	...	...	...	...	...	...	...	...	...
L2C3	...	...	...	...	...	...	...	...	...
L3C1	...	...	...	...	...	...	...	...	...
L3C2	...	...	...	...	...	...	...	...	...
L3C3	...	...	...	...	...	...	...	...	...

DR 1 – Matrice d'adjacence associée quadrillage d'échauffement