



# Projet IA

---

*Déplacement d'un pion via le plus court chemin*

Étudiants	Laurie Tissier-Verse Juliette Sta Quentin Périé
Projet	Intelligence Artificielle
Date de lancement	07 / 03 / 2018
Référence du document	IA_Salotti_Tissier-Verse_Sta_Périé

# Sommaire

---

<b>Sommaire</b>	<b>2</b>
<b>Introduction</b>	<b>3</b>
<b>I - Modélisation</b>	<b>4</b>
Représentation graphique	4
Cas A : Case initiale et case finale	5
Cas B : 4 cases	7
<b>II - Tests</b>	<b>8</b>
Gestion des erreurs d'entrée	8
Cas A : Case initiale et case arrivée	9
Test A.1	9
Test A.2	9
Test A.3	10
Exemple de TreeView sur le Test A.3 :	10
Test A.4	11
Cas B : Cas avec 4 checkpoints où passer	12
Test B.1	12
Test B.2	12
Test B.3	13
<b>Conclusion</b>	<b>14</b>

# Introduction

---

Le second semestre de la 2ème année à l'ENSC se focalise sur la découverte des notions d'intelligence artificielle, de la façon de l'appréhender, de réfléchir, au codage de programme intelligents, permettant de résoudre de nombreux problèmes.

Un projet est ainsi mis en place afin de nous plonger dedans. Il s'agit dans un premier temps de faire un programme qui calcul puis affiche le chemin le plus court pour se déplacer d'un point à un autre au sein d'une grille prédéfini. La difficulté réside dans le fait qu'il y ai des murs à éviter. Dans un second temps le chemin ne va plus d'un point à un autre mais doit parcourir une boucle à partir d'un point initial en passant par 4 checkpoints prédéfinis.

Pour réaliser ce projet, il nous est fourni en langage C# l'algorithme du plus court chemin A\* ainsi qu'une classe *GenericNode* composé de méthodes abstraites à compléter dans les classes que nous créerons :

- *Node* pour les noeuds dans le cas d'un chemin simple
- *Node4* pour les noeuds dans le cas d'un chemin devant passer par 4 checkpoints

L'algorithme A\*, extension de l'algorithme de Dijkstra, permet ainsi de trouver le plus court chemin pour répondre à notre problématique. Il faut en revanche lui fournir des données qu'il nous incombe de calculer à l'aide d'heuristiques à bien définir.

Le projet s'est ainsi décomposé en deux grandes parties. La première consistait en la création d'une interface utilisateur et la modélisation du problème avec une réflexion sur les heuristiques à mettre en place et leur implémentation. Quant à la seconde, elle se composa des différents tests à effectuer, permettant ainsi de modifier, améliorer puis valider notre travail.

# I - Modélisation

---

Pour pouvoir utiliser l'algorithme A\* et le programme fourni, il nous faut modéliser puis écrire le code nécessaire. Pour cela une décomposition en deux parties a été faite : d'une part l'interface avec l'utilisateur, le formulaire, le retour sous forme graphique, de treeview etc. et d'une autre part l'implémentation des classes et méthodes nécessaires.

## Représentation graphique

L'interface entre l'utilisateur et le projet technique se fait via une application Windows Form. L'interface graphique comprend une représentation graphique : Grille ou TreeView, une partie pour trouver le plus court chemin entre 2 points, et une partie pour trouver le plus court chemin qui relie un point de départ à 4 checkpoints (ou moins si on entre deux fois les mêmes coordonnées).

Un bouton permet de réinitialiser la grille (la matrice qui stocke les cellules accessibles ou non). Les cases où l'on peut se déplacer sont en blanc; celles où l'on ne peut pas, en noir.

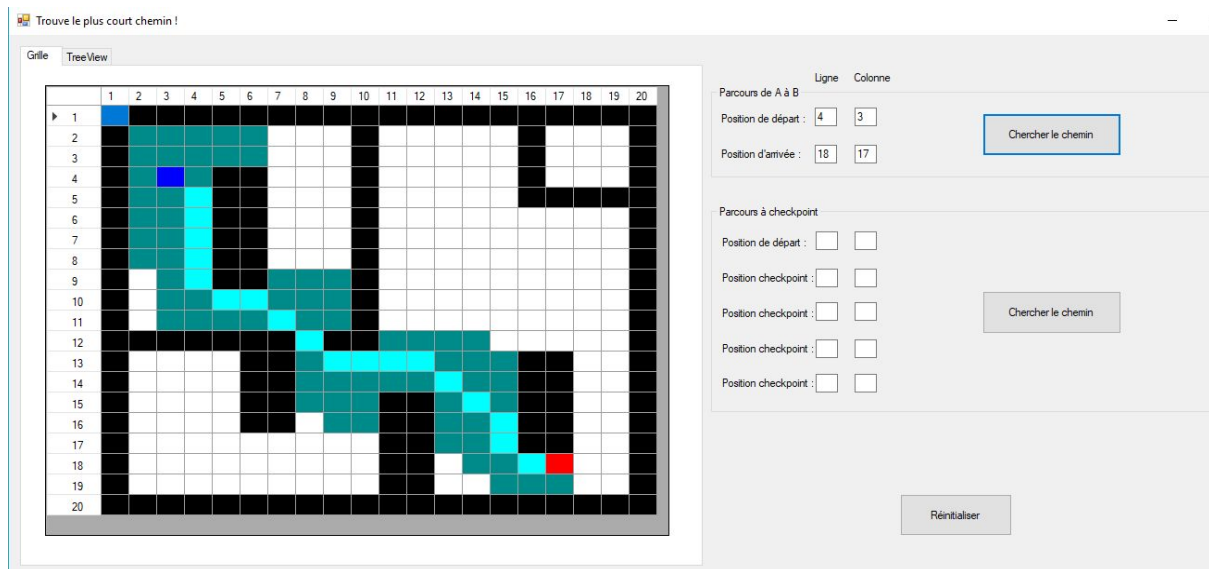
Quand on entre les coordonnées des points puis que l'on cherche le plus court chemin, celui-ci s'affiche en cyan. Le départ est alors en bleu et l'arrivée en rouge (*exemple page suivante*). Les checkpoints sont en verts. Les couleurs sont stockées dans une matrice par des entiers.

Valeur stockée	Couleur correspondante	Case correspondante
0	Noir	Mur
1	Blanc	Case parcourable
2	Bleu	Case initiale
3	Vert	Checkpoints
4	Rouge	Case finale
5	Cyan	Case parcourue lors du chemin le plus court
6	DarkCyan	Case explorée par l'algorithme A*

Le formulaire est notre programme de référence est stocke les variables suivantes :

- La cellule de départ,
- La cellule d'arrivée,
- La cellule de départ si on veut passer par les checkpoints,
- Les checkpoints,
- La matrice de cellules.

Ces variables sont définies publiques et statiques car elles doivent être accessibles partout dans la solution et sont uniques. Il n'y a qu'une matrice de cellules permettant de savoir où sont les cellules inaccessibles. La classe *Node* en a besoin pour connaître la liste de ses successeurs - les cases accessibles qui l'entourent.



Capture 1 : Aperçu du Windows Form

## Cas A : Case initiale et case finale

La classe dérivée de *GenericNode* est ici appelé *Node*. Elle possède comme attribut *x* et *y* qui correspondent respectivement à la ligne et à la colonne de la case où se situe le noeud.

Les fonctions de *Node* prennent en compte la case de départ et la case d'arrivée que rentre l'utilisateur.

Le noeud est un noeud final s'il correspond aux coordonnées de la case d'arrivée entrées précédemment par l'utilisateur.

Le coût de l'arc, qui représente le coût pour accéder au noeud suivant est la distance euclidienne. Puisqu'ici la distance entre deux cases côte à côte (verticalement ou horizontalement) est de 1, la distance pour deux cases en diagonale sera de  $1 * \sqrt{2} = \sqrt{2}$ .

On peut ainsi représenter les coûts d'arc comme suit :

$\frac{\sqrt{2}}{2}$	1	$\frac{\sqrt{2}}{2}$
1	0	1
$\frac{\sqrt{2}}{2}$	1	$\frac{\sqrt{2}}{2}$

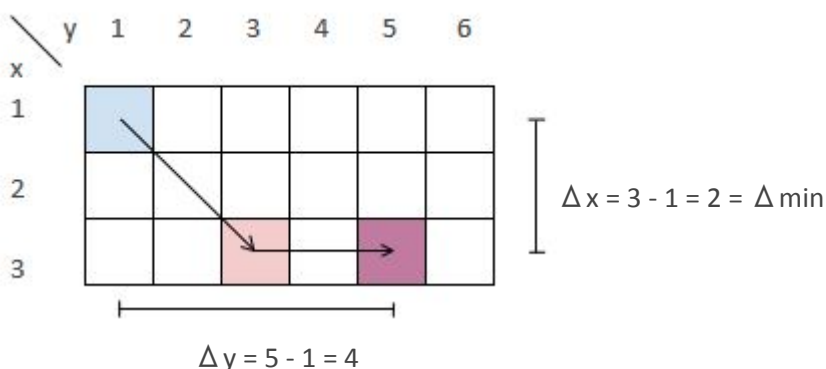
Pour calculer le chemin le plus court, il faut regarder tous les chemins possibles qui commencent par les cases adjacentes. Dans la grille, les cases impossibles sont associées au 0. On regarde donc si la case suivante est bien dans la matrice (coordonnées existantes) et si sa valeur est différente de 0 (case parcourable).

Concernant l'heuristique, on calcule le chemin minimum qu'il reste à parcourir jusqu'à la case d'arrivée. L'idéal est un chemin suivant une diagonale puis un trajet vertical ou horizontal.

Pour cela on récupère les écarts (horizontal et vertical) en valeur absolue entre la case d'arrivée et le noeud actuel :  $\Delta x$  et  $\Delta y$ .

On cherche ensuite qui de  $\Delta x$  et  $\Delta y$  est le plus petit : on appellera cet écart  $\Delta \min$ .  $\Delta \min$  est donc la distance à parcourir en diagonale pour se situer dans la ligne ou la colonne du noeud à atteindre. On ajoute donc au coût  $\sqrt{2} * \Delta \min$ .

En enlevant  $\Delta \min$  aux deux écarts, l'un des deux vaut zéro. Visuellement, on peut imaginer un nouveau noeud qui se situerait sur la même ligne ou colonne que notre noeud d'arrivée (voir le schéma suivant). Il reste donc à ajouter le nombre de case les séparant.



Pour cet exemple, la case bleue représente la case de départ (1,1) et la case violette celle d'arrivée (5,3). L'écart entre les deux cases verticalement vaut  $3-1=2$  et l'écart horizontalement vaut  $5-1=4$ . Or on a  $2 < 4$ . Donc le coût intermédiaire vaut  $\sqrt{2} * 2$ . Visuellement on arrive à la case en rose (3,3). Les écarts sont à nouveau calculés entre la case rose et violette : verticalement il vaut 0 et horizontalement il vaut 2.

Le coût final vaut alors le coût intermédiaire plus 2. C'est à dire ici :  $2\sqrt{2} + 2$ .

### Cas B : 4 cases

La classe dérivée de *GenericNode* est ici appelé *Node4*. Comme *Node*, elle possède comme attribut *x* et *y* qui correspondent respectivement à la ligne et à la colonne de la case où se situe le noeud que nous regardons. Elle a également 4 booléens, lui permettant de savoir si dans ses prédécesseurs une cellule était un checkpoint.

Lors de l'exploration des solutions par l'algorithme A\*, chaque noeud va instancier ses noeuds voisins dans une liste (*GetListSucc*). Lors de leur instanciation, la valeur des booléens (*checkpoint passé*) est transmise aux successeurs.

On parcourt dans cette même méthode les successeurs. Si celui-ci est un checkpoint le booléen concerné passe à vrai (et sera ensuite transmis aux successeurs).

Le noeud est déterminé comme un noeud final s'il correspond à la case initiale et si ses booléens sont tous vrais. C'est-à-dire si avant d'arriver à ce noeud tous les noeuds correspondants aux checkpoints ont été visités.

Le coût de l'arc ne change pas, la distance au prochain noeud se calcule toujours en distance euclidienne. De même, les caractéristiques de la grille ne changent pas.

Pour le calcul de l'heuristique, nous avons distingué deux cas :

- Si tous les intermédiaires ont été visités, il ne reste plus qu'à rejoindre la case initiale. On effectue alors la même opération qu'avec *Node*.
- S'il reste au moins un intermédiaire non visité : on cherche le point intermédiaire non visité le plus loin en distance euclidienne auquel on ajoute le coût pour revenir à la case initiale. En effet, pour visiter tous les points intermédiaires il faudra au moins aller jusqu'au plus loin puis revenir à la case de départ (cas des checkpoints tous alignés, voire au même endroit).

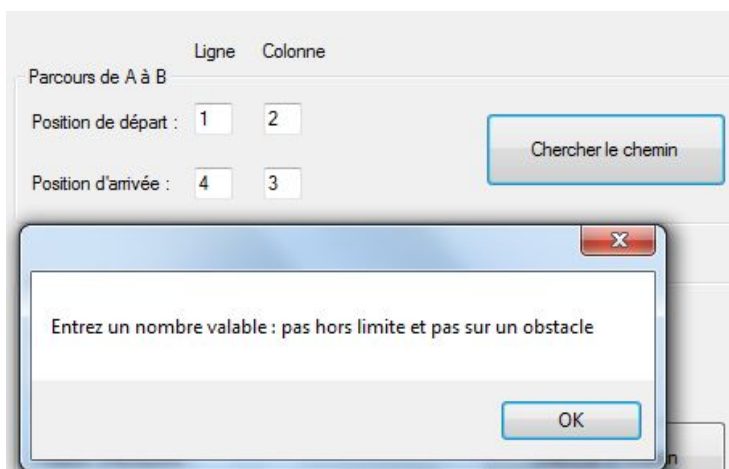
## II - Tests

Les heuristiques étant choisies et le programme ayant été complété, il faut ensuite mettre en place les tests nécessaires à la vérification du bon fonctionnement du programme. Pour cela on dispose d'une suite de tests, proposée dans le sujet du projet.

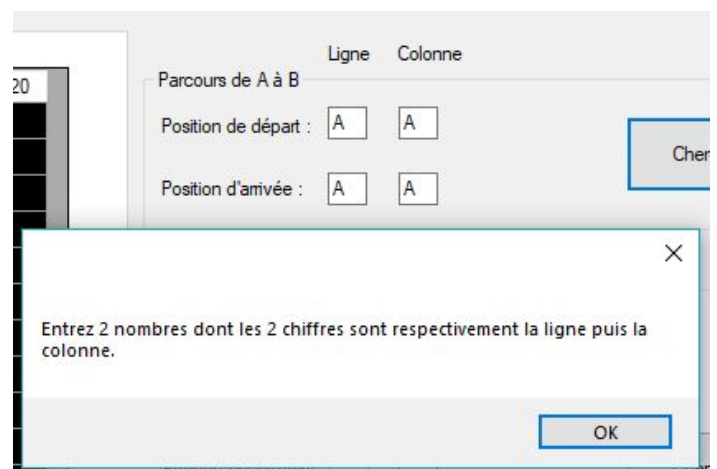
### Gestion des erreurs d'entrée

Avant de tester les recherches de chemins et le fonctionnement même du programme, nous avons testé et validé la gestion des erreurs d'entrées. Ainsi, en cas de saisie incorrecte ou impossible, l'utilisateur est prévenu et est invité à recommencer l'opération.

Cette situation peut être le résultat d'une coordonnée de case qui n'existe pas (Exemple 1 : hors limite de la grille ou dans une cellule inaccessible) ou bien lorsqu'une coordonnées n'est pas entrée (Exemple 2 : la coordonnée de la ligne de la case d'arrivée).



Exemple 1 : Nombre hors limite



Exemple 2 : Absence de nombre



Parcours de A à B

Ligne Colonne

Position de départ :

Position d'arrivée :

Entrez un nombre dont les 2 chiffres sont respectivement la ligne puis la colonne.

Exemple 3 : Absence d'une coordonnée

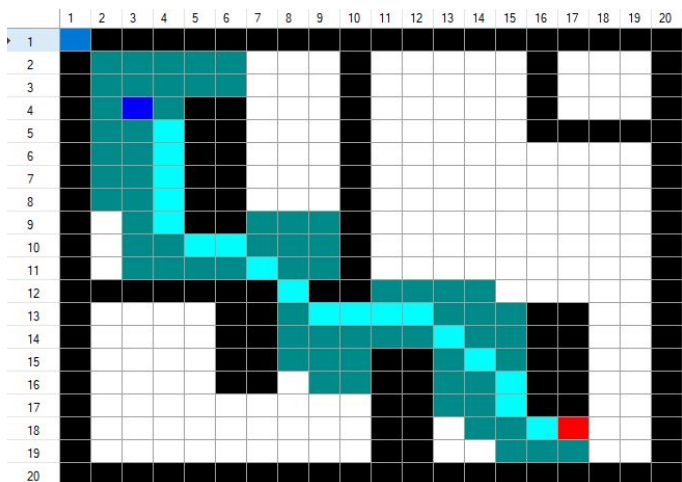
### Cas A : Case initiale et case arrivée

Pour chaque cas fonctionnel nous avons en bleu foncé la case de départ, en rouge celle d'arrivée et en cyan le chemin le plus court. Pour les cas qui ne marchent pas, dans le deuxième onglet on peut accéder au tree view.

Test A.1

Départ → case de coordonnées : (4,3)

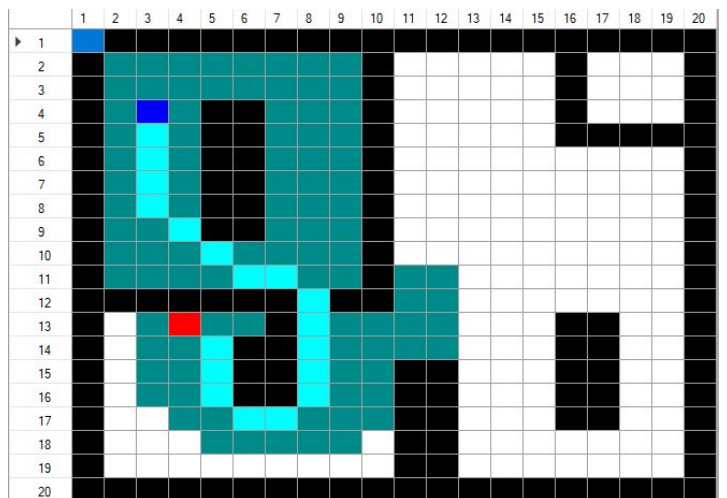
Arrivée → case de coordonnées : (18,17)

Test A.1 : résultat

Test A.2

Départ → case de coordonnées : (4,3)

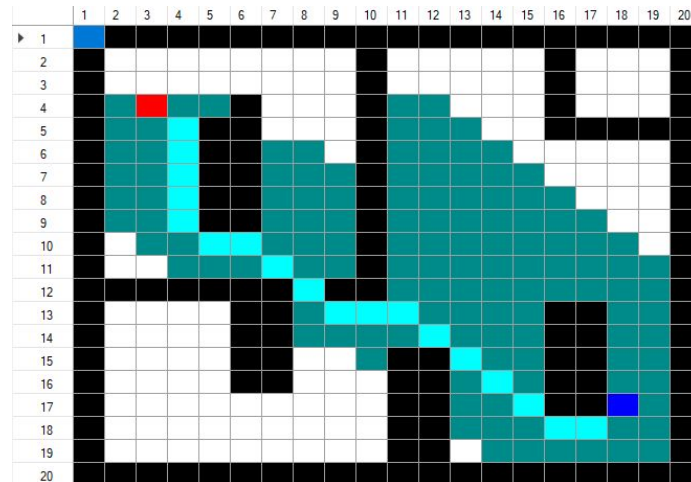
Arrivée → case de coordonnées : (13,4)

Test A.2 : résultat

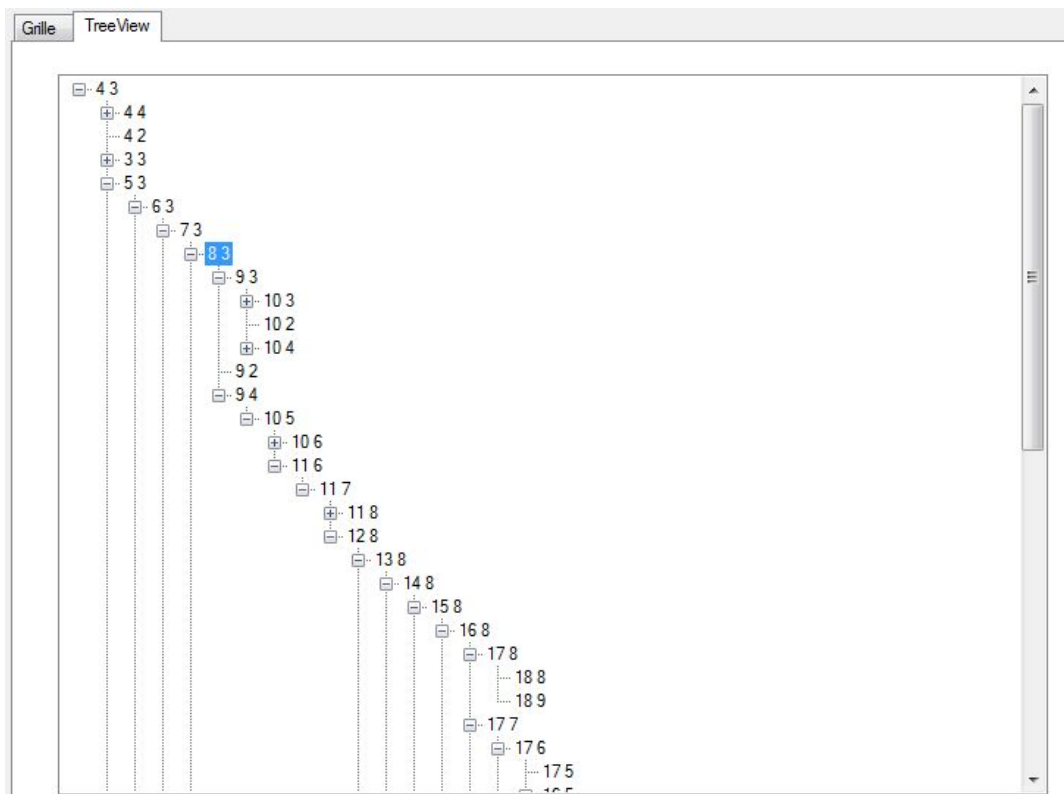
## Test A.3

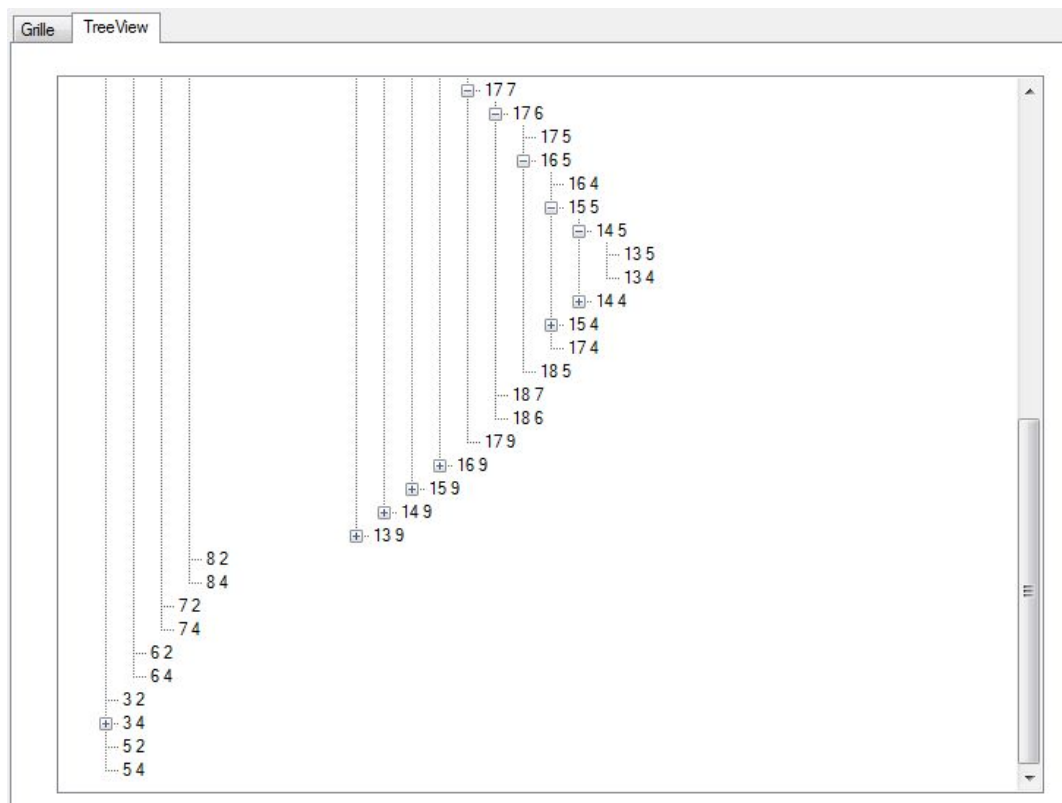
Départ → case de coordonnées : (17,18)

Arrivée → case de coordonnées : (4,3)

Test A.3 : résultat

Exemple de TreeView sur le Test A.3 :



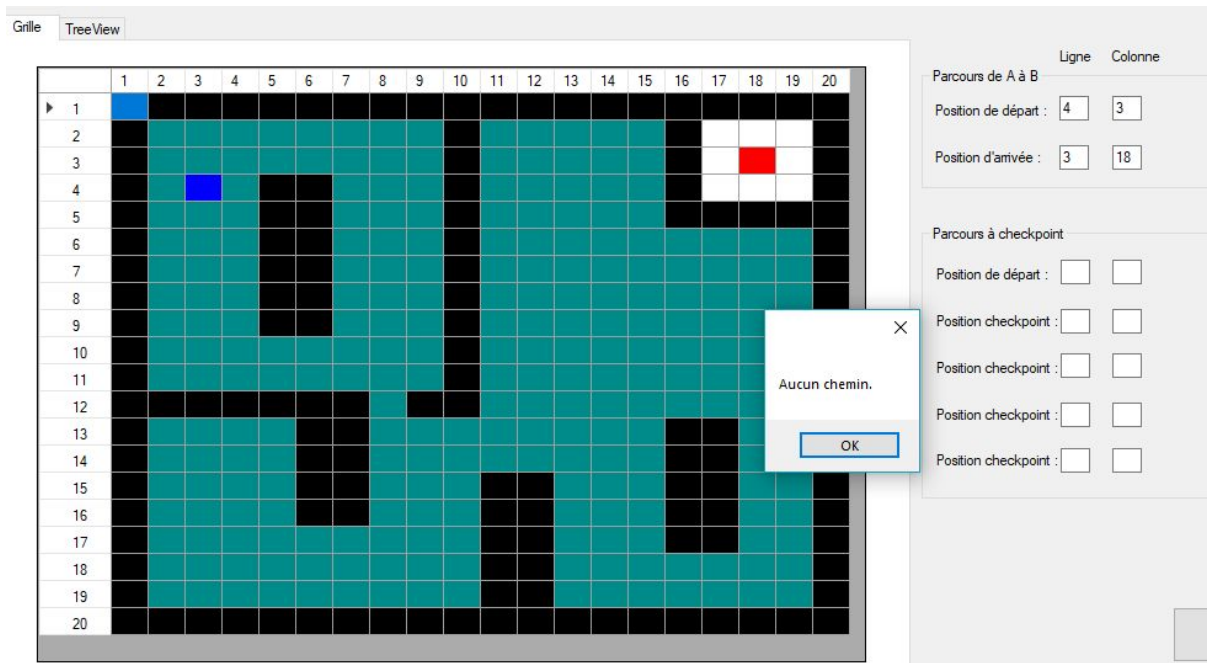
TreeView Test A.3

## Test A.4

Départ → case de coordonnées : (4,3)

Arrivée → case de coordonnées : (3,18)

Cas où le chemin n'est pas possible.


Test A.4 : résultat

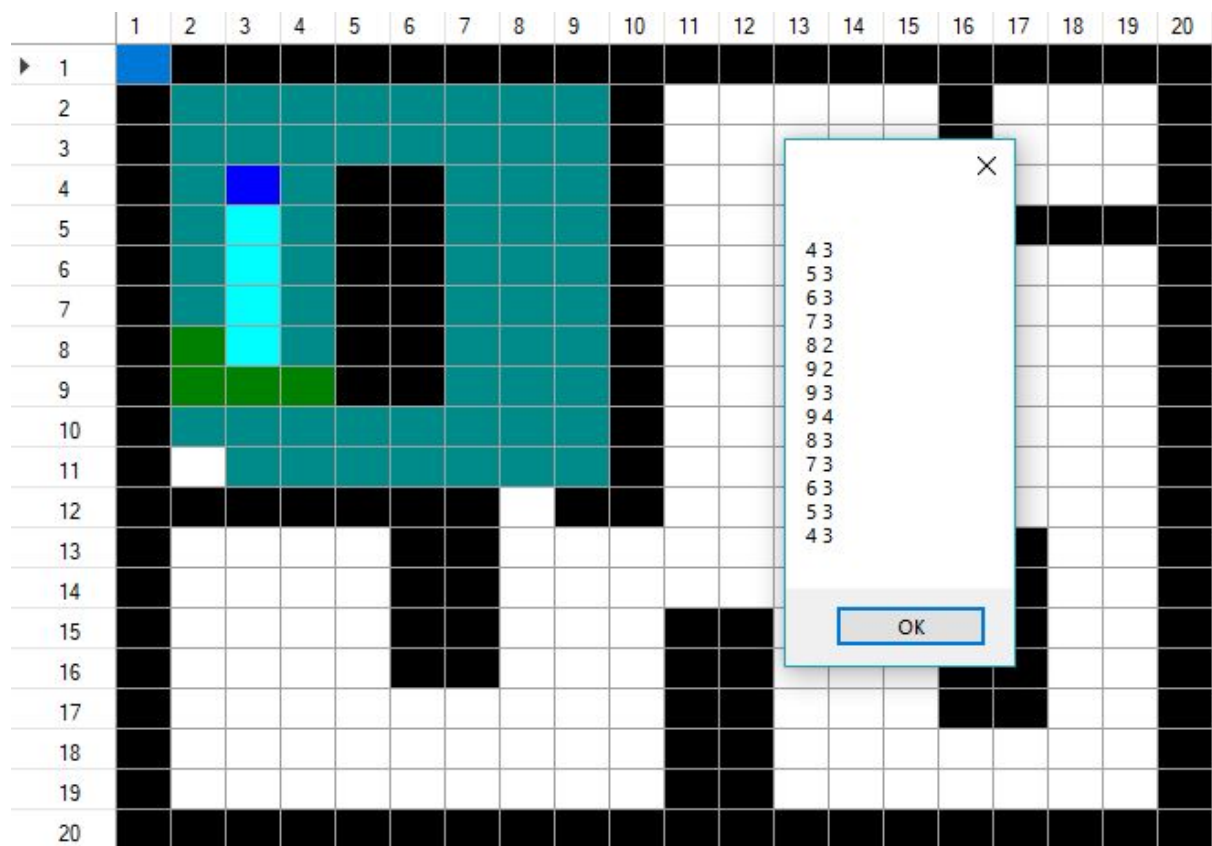
## Cas B : Cas avec 4 checkpoints où passer

Pour comprendre par comment se déroule le chemin (coordonnées + ordre), nous avons décidé d'afficher les coordonnées de chaque cellules du chemin dans l'ordre de passage.

### Test B.1

Départ → case de coordonnées : (4,3)

Checkpoints → cases de coordonnées : (8,2) ; (9,2) ; (9,3) ; (9,4)



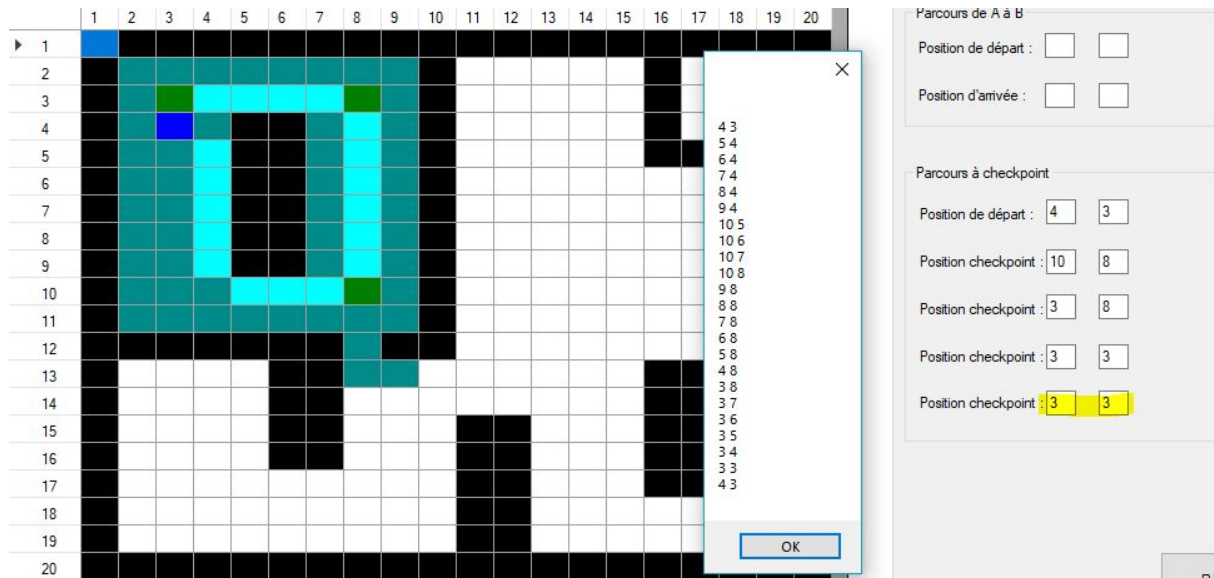
Test B.1 : résultat

### Test B.2

Départ → case de coordonnées : (4,3)

Checkpoints → cases de coordonnées : (10,8) ; (3,8) ; (3,3)

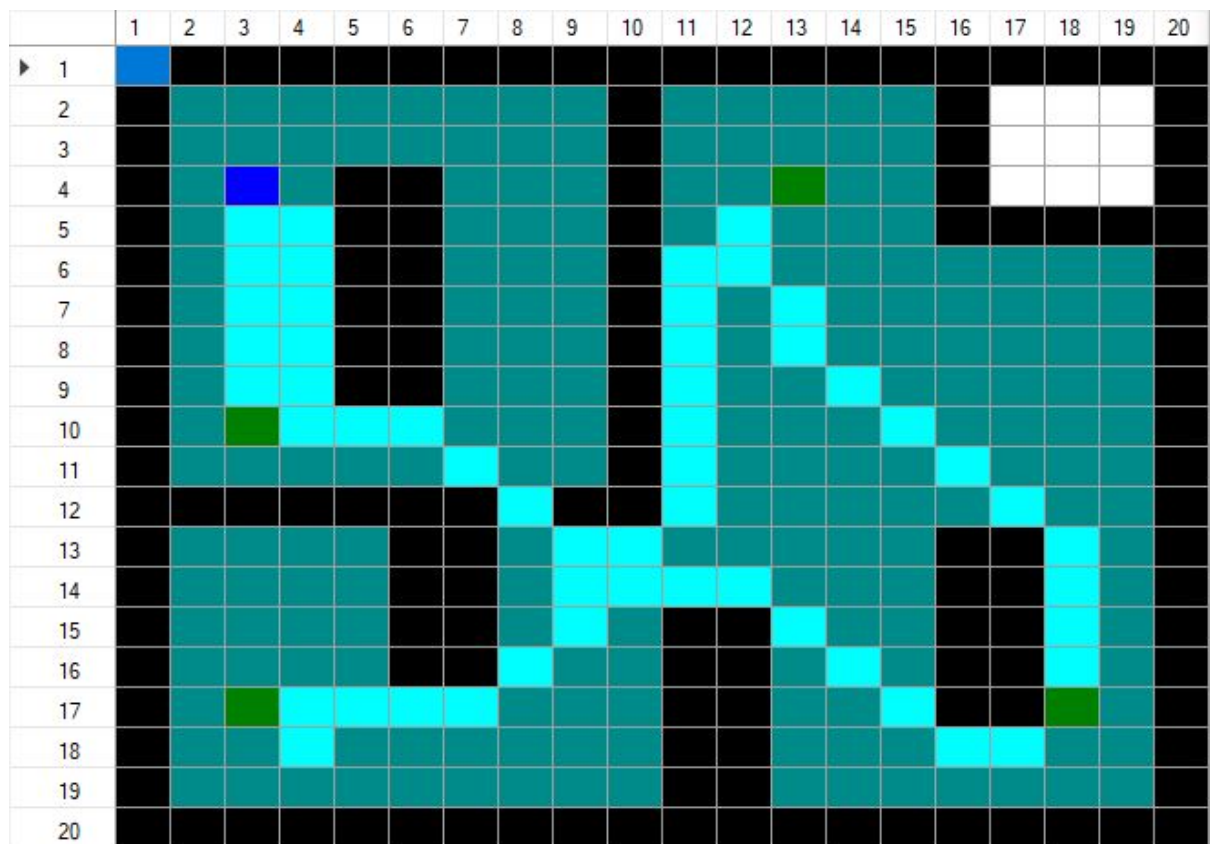
Afin de mettre moins de 4 checkpoints il faut entrer 2 fois les mêmes coordonnées (surligné sur la capture d'écran).

Test B.2 : résultat

## Test B.3

Départ → case de coordonnées : (4,3)

Checkpoints → cases de coordonnées : (10,3) ; (17,3) ; (4,13) ; (17,18)

Test B.3 : résultat

# Conclusion

---

Ainsi, ce projet nous aura permis d'aborder les notions d'intelligence artificielle vues en cours avec notamment l'exploitation de l'algorithme A\*. Si celui-ci était déjà implémenté, il était nécessaire de bien le comprendre afin de faire le choix d'heuristiques viables et pouvoir générer des classes *Node* et *Node4* efficaces et fonctionnelles.

La mise en forme du résultat s'est également avérée être d'une grande importance. La visualisation de la sortie d'algorithme permet en effet un gain de temps considérable lors d'un tel projet pour pouvoir voir ce qui fonctionne ou ne fonctionnent pas.

Finalement, on peut ajouter que des améliorations sont possibles, notamment pour ce qui est du calcul de coût via notre heuristique qui pour l'instant calcule un coût minimal mais non exact. Il est donc possible de faire mieux.

La classe *Node4* pourrait être également généralisée à *NodeX* avec X checkpoints. Les heuristiques prendraient alors d'autant plus d'importance. Il s'agirait en effet de faire quelque chose de plus précis mais sans pour autant que cela demande trop de calculs.

De même, la grille ici prédéfinie en dur dans le code pourrait avoir la possibilité d'être modifiée, agrandie etc. en demandant la taille de départ puis en définissant les cases non accessibles en cliquant dessus par exemple.