Projet IA : Navigation d’un voilier

Parize Antoine - Vallcaneras Julie

Groupe 1

Année 2020 - 2021

Table des matières

I - Introduction

Ce projet d'introduction à l'intelligence artificielle s'inscrit dans le domaine de la résolution de problème par la recherche du plus court chemin à emprunter. En effet, le but de ce projet est de créer un algorithme capable de déterminer le chemin qu'un voilier doit emprunter pour minimiser son temps de trajet, en prenant en compte la direction du vent, sa force, et la direction du bateau lorsqu'il navigue.

Pour résoudre ce problème nous nous sommes appuyé sur l'algorithme A*, que nous décrirons plus tard dans ce rapport. Cet algorithme permet de parcourir les noeuds d'un graph pour trouver le plus court chemin, c'est à dire le chemin nécessitant le coup le plus bas pour atteindre la destination finale.

Trois cas devaient être traités au cours de ce projet, mais nous verrons que l'algorithme créé

est capable de généraliser la recherche du plus court chemin quel que soit le point de départ et d'arrivée voulu.

II - Gestion de projet

Au cours de ce projet, nous avons fonctionné en binôme. Nous nous sommes réparti les tâches de telle sorte à ce que l'un travaille sur l'architecture du programme, l'implémentation des successeurs et les autres fonctions mineures, et l'autre travaille sur la partie graphique et l'implémentation de l'heuristique.

Fonctionnalité	Responsable
Modélisation du problème	Antoine et Julie
Création de l'architecture logiciel	Julie
Implémentation classe Vent	Julie
Implémentation IsEqual	Julie
Implémentation GetListSucc	Antoine
Implémentation GetArcCost	Julie
Implémentation CalculHCost	Antoine
Implémentation ToString	Antoine
Implémentation du winforms	Antoine
Implémentation affichage du chemin parcouru par le voilier	Julie
Optimisation Successeurs/heuristique	Antoine
Rédaction du rapport	Antoine et Julie

III - Résolution du problème

A - Problématique de l'exercice

Le problème posé par le sujet est le suivant : un voilier doit se rendre d'un point P1 à un point Pf. Ce voilier se déplace plus ou moins rapidement en fonction de la direction du vent et de sa force, et de la direction du voilier lui-même. Notre objectif était donc de créer un algorithme permettant d'optimiser le trajet du voilier, c'est à dire trouver le plus court chemin entraînant le moins de coût lors du trajet. Le coût du trajet étant ici exprimé en temps de trajet (en heures). Notre algorithme doit donc être capable de prendre en entrée les coordonnées des points de départ P1 et d'arrivée Pf, un cas à traiter (e.g, a, b ou c) et déterminer le plus court chemin minimisant le temps de trajet du voilier.
Pour résoudre ce problème nous avons utilisé l'algorithme A* décrit dans la section suivante.

Ce projet impliquait aussi la réalisation d'une interface graphique permettant d'utiliser notre programme et d'afficher efficacement et de manière compréhensible nos résultats. Cette interface a été réalisée en windows form.

B - Algorithme A*

Comme nous l'avons vu plus haut, notre problème à résoudre est un problème de recherche du plus court chemin. Ce problème peut être modélisé sous forme de graph. En effet, chaque position sur la carte est un noeud avec des coordonnées X et Y associées. Chaque trajet entre deux positions (i.e, noeuds) est une arête qui possède un coût. Par conséquent, nous avons pu nous appuyer sur l'algorithme A* qui permet de parcourir un graph et de retourner le chemin allant du noeud initial P1 au noeud final Pf en minimisant le coût du trajet, c'est à dire ici le temps mis en heures par le voilier pour rallier le point final. Cet algorithme s'appuie notamment sur une heuristique, un calcul permettant d'estimer la distance séparant le noeud que l'on est en train d'évaluer avec le noeud final. Utiliser une heuristique efficace permet à l'algorithme d'être plus rapide, c'est pourquoi nous avons essayé d'implémenter l'heuristique la plus pertinente pour notre problème tout en prenant en compte certains compromis de temps de calcul et de précision dans le chemin trouvé par l'algorithme.

C - Liste des successeurs

Dans le présent problème on peut imaginer différentes sortes d'explorations de graph, chacune pouvant apporter des avantages et des inconvénients. Ici en effet, notre voilier se déplace de positions en positions en effectuant des trajets de maximum 10 kilomètres. Nous devons donc prendre en compte cette contrainte dans l'exploration des noeuds voisins. Le problème ici est donc de proposer un pavage permettant l'exploration des noeuds proches du noeud courant tout en excluant les noeuds à plus de 10 kilomètres. Par conséquent, plusieurs pavages différents ont retenu notre attention : un pavage en carré de 1, en diagonal de 1, et enfin en carré de taille variable.

En plus de ces différents pavages, nous avons ajouté une fonctionnalité permettant à l'algorithme d'ajouter à la liste des noeuds ouverts le noeud final, si ce dernier est à 10 kilomètres ou moins du noeud courant. Cette fonctionnalité nous semblait utile car il est envisageable que l'algorithme ne trouve jamais de solution car il n'explorerait pas les noeuds du graph jusqu'au noeud final. Ajouter cette fonctionnalité permettrait donc dans une moindre mesure d'assurer une solution à notre recherche du plus court chemin avec l'algorithme A*

1 - Pavage en carré de 1

Ce pavage nous a semblé le plus simple et le plus classique. C'est un pavage style "démineur", où l'on regarde l'ensemble des 8 noeuds adjacents directement au noeud courant en carré autour de lui. Comme nous prenons en compte les noeuds diagonaux au noeud courant en ajoutant ou retranchant 1 à leurs coordonnées, la distance entre deux noeuds en diagonale est $\sqrt{((x+1)-x)^2+((y+1)-y)^2} = \sqrt{2}\,km$ et pas 1 km. Cependant, cela nous permet d'obtenir des noeuds ayant des coordonnées non décimales, ce qui réduit a priori le nombre de noeuds potentiels à explorer. En effet, en interdisant des coordonnées décimales ici, on empêcherait le programme de potentiellement explorer indéfiniment le graph des positions, car l'exploration des successeurs pourrait créer une infinité de noeuds ne se recoupant jamais.

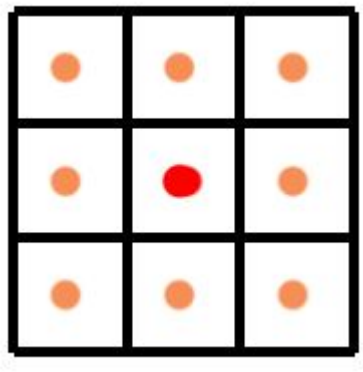

Figure 1 : Exemple d'un pavage carré de 3. Le côté c du carré est donc égal à 7. Le point rouge indique le noeud courant, et les points oranges les noeuds successeurs. On notera l'exclusion des noeuds sur la même ligne ou colonne que le noeud courant pour accélérer les calculs.

2 - Pavage diagonal de 1

Après avoir observé les résultats obtenus avec le pavage carré, nous nous sommes rendu compte que le programme utilisait bien plus souvent un déplacement en diagonale qu'un déplacement latéral. Nous avons donc implémenté un pavage ne proposant que les noeuds en diagonale de 1 du noeud courant. Ceci a pour effet d'évincer des noeuds successeurs latéraux parfois inutiles dans le calcul du plus court chemin.

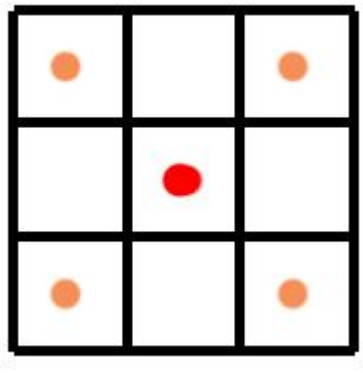

Figure 2 : Exemple d'un pavage diagonal. Le point rouge indique le noeud courant, et les points oranges les noeuds successeurs.

3 - Pavage en "carré" variable

Ce dernier pavage permet d'obtenir en successeurs l'ensemble des noeuds dans un carré autour du noeud courant. Ce carré est de côté $c = p \times 2 + 1$, p étant compris dans l'intervalle [1 ; 8]. Nous avons cependant choisi d'exclure volontairement l'ensemble des

noeuds se trouvant sur la même ligne ou colonne que le noeud courant pour diminuer le temps de calcul. Dans cette même optique nous avons choisi de ne garder que les noeuds dont les coordonnées étaient paires. De ce fait, notre algorithme a été rendu bien plus rapide et efficace (voir Figure 3).
L'utilisateur peut choisir préalablement au lancement de l'algorithme la taille du carré des successeurs. Cette taille peut prendre une valeur dans l'intervalle [1 ; 8].

Ce pavage nous permet de créer des déplacements en diagonales minimisant la distance à parcourir pour le bateau. On donne en fait la possibilité au programme d'utiliser des trajectoires longues et plus ***lisses***. Ceci nous semblait d'autant plus pertinent que la vitesse du bateau dépend de la direction de ce dernier et de la direction du vent. En permettant à l'algorithme d'utiliser de telles trajectoires, nous supposions que le chemin trouvé serait plus optimisé en termes de distance et de temps de parcours. En effet, une trajectoire rectiligne semble instinctivement plus courte en termes de temps de parcours comparé à une trajectoire qui ne l'est pas (voir Figure 4).

Cependant, en faisant cela on permet au programme de regarder un grand nombre de successeurs à chaque noeuds. On peut donc imaginer que le temps de calcul est nettement plus long avec ce pavage.

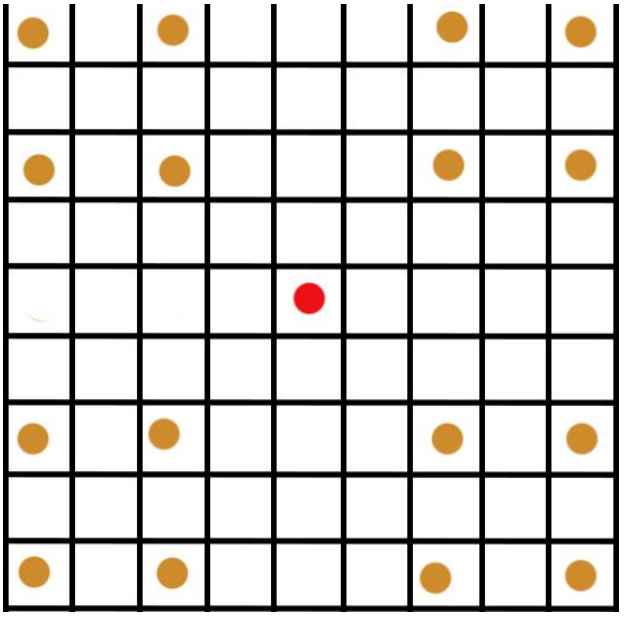

Figure 3 : Exemple d'un pavage carré de 4. Le côté c du carré est donc égal à 9. Le point rouge indique le noeud courant, et les points dorés les noeuds successeurs. On notera l'exclusion des noeuds sur la même ligne ou colonne que le noeud courant pour accélérer les calculs. Les noeuds à coordonnées X ou Y impaires sont eux aussi exclus pour accélérer les calculs

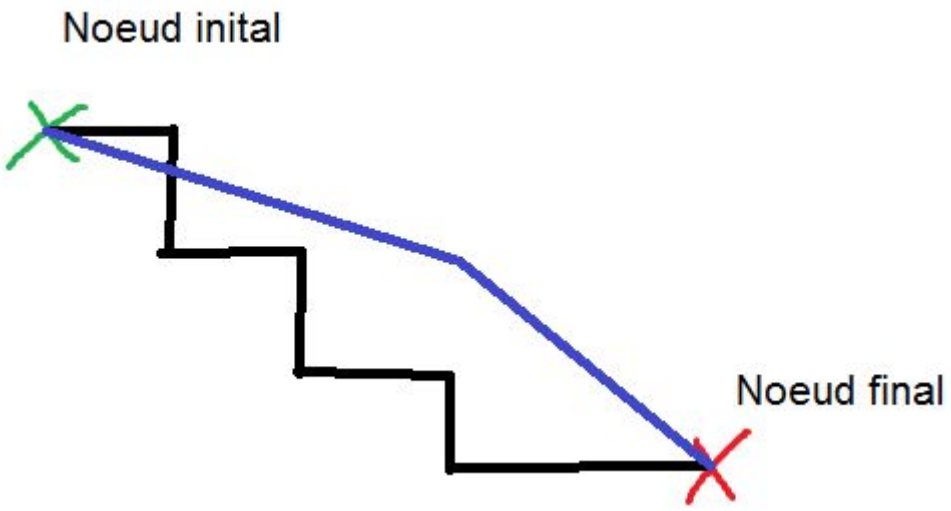

Figure 4 : Illustration de deux chemins différents entre un noeud initial et final. Le chemin en noir est une trajectoire rectiligne sans diagonale. Le chemin en bleu est obtenu grâce à des diagonales avec un écart fort entre les noeuds, ce qui résulte en une trajectoire plus lisse.

D - Heuristiques Potentielles

1 - Principe général de nos heuristiques

Dans chacune de nos heuristiques nous avons fait en sorte que la valeur calculée soit un **temps de trajet** entre le noeud courant et le noeud final. En utilisant un temps de trajet, nous calculons une valeur qui reste à la même échelle que le coût normal de trajet entre deux noeuds. Utiliser seulement une distance entre le noeud courant et le noeud final paraît de prime abord pertinent. Cependant, l'heuristique calculée n'est alors pas dans le même ordre de grandeur que le coût entre deux noeuds. Il en résulte donc un "écrasement" du coût entre deux noeuds par l'heuristique et l'algorithme ne se base plus que sur cette dernière pour prendre ses décisions. On obtient donc un algorithme très rapide mais qui calcule une solution sous-optimale voire médiocre.

2 - Compromis vitesse de calcul - précision

Choisir une heuristique appropriée dans ce projet s'est révélé être un problème épineux. En effet, le problème auquel nous faisons face est en soi extrêmement complexe, ce qui rend difficile la détermination de la meilleure heuristique.
Nous nous sommes rendu compte pendant ce projet que le problème n'était pas de définir une heuristique fonctionnelle, mais plutôt définir la meilleure heuristique par rapport au problème à traiter, avec toute la complexité qui lui est inhérente. Dans ce problème du voilier, nous sommes face à un graph dont les coûts entre les arcs ne sont pas uniformes d'abord, mais qui de plus changent en fonction de la position du noeud dans le graph (i.e,

direction du vent en fonction de X et Y) et de comment on le parcourt (direction du vent et du bateau). De plus, la possibilité d'explorer des noeuds situés à moins de 10km suppose que l'on a une très large possibilité de parcours du graph, ce qui rajoute de la complexité à notre problème.

Sachant que le temps de trajet de notre voilier serait optimisé en utilisant des trajectoires lisses, c'est à dire en permettant de regarder beaucoup de noeuds proches comme successeurs, on voudrait trouver une heuristique permettant un calcul assez rapide de la solution tout en étant efficace. En résumé, la solution devrait être trouvée vite et le temps de trajet doit être le plus court possible. Ce compromis devient très vite complexe car la solution la plus précise semble devoir être trouvée avec un parcours exhaustif du plus de trajectoires possible, ce qui augmente rapidement le temps de calcul nécessaire pour trouver une solution.

Comme nous n'avions pas en tête une solution à cette heuristique mais plusieurs, nous avons essayé d'implémenter plusieurs heuristiques différentes afin de les comparer.

3 - Temps basé sur une distance euclidienne

Comme nous l'avons évoqué auparavant, un noeud peut avoir en successeur n'importe quel noeud situé à moins de 10 kilomètres. Ceci implique que la trajectoire que le bateau peut prendre peut être de n'importe quel angle, si nous donnons à l'algorithme la possibilité de le faire. On peut par conséquent envisager une heuristique calculant le temps de trajet direct entre le noeud courant et le noeud final. En d'autres termes, l'heuristique serait un temps de trajet basé sur la distance euclidienne entre le noeud courant et le noeud final. Nous avons choisi de calculer cette heuristique en donnant au bateau sa vitesse maximale peu importe sa position ou sa direction.

L'heuristique correspondante est calculée comme suit : $h_1 = \frac{\sqrt{(X_i - X_f)^2 + (Y_i - Y_f)^2}}{45}$,
45 étant la vitesse maximale que peut atteindre le bateau dans les meilleures conditions possibles.

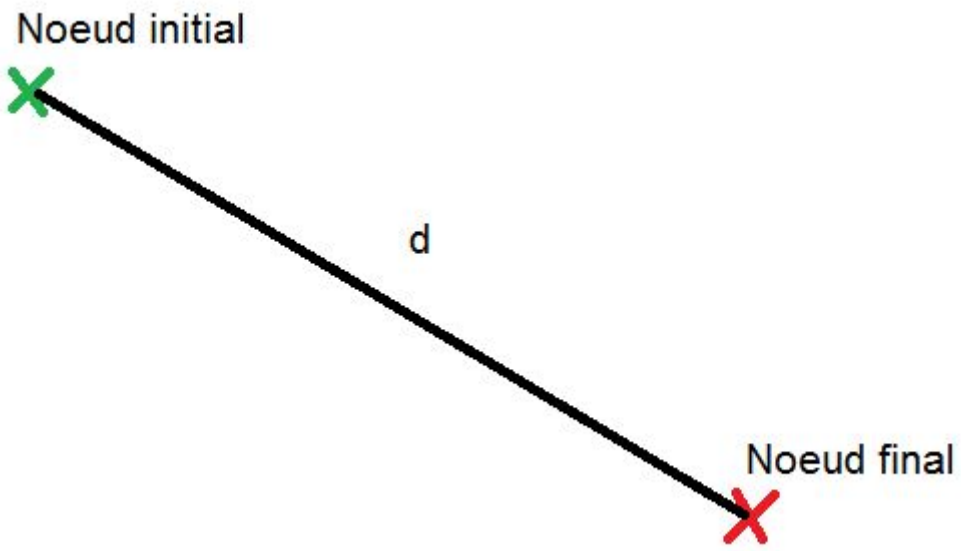

Figure 5 : Illustration de la distance euclidienne d entre le noeud initial et le noeud final.

4 - Temps basé sur une distance de Manhattan avec diagonales

Notre pavage initial étant un pavage en carré de 1, notre algorithme ne pouvait pas bénéficier de trajectoires lisses pour parcourir le plan. Nous avons donc essayé d'ajuster notre heuristique en considérant cette contrainte.

Par nos recherches nous avons trouvé qu'utiliser une "octile distance" serait particulièrement approprié ici. Cette distance est en fait une distance de Manhattan qui admet des diagonales, et dont le coût associé à ces diagonales n'est pas 1 mais $\sqrt{2}$ (Zhang, 2016). Une distance de Manhattan classique n'aurait pas été appropriée car le pavage en carré permettait une exploration en diagonale, ce qui est contraire au principe même de la distance de Manhattan.

L'heuristique obtenue grâce à l'octile distance est calculée comme suit :

$$h_2 = D_1 \times (dx + dy) + D_2 \times min(dx, dy)$$

avec $D_1 = 1$, $D_2 = \sqrt{2}$, $dx = \left|X_f - X_i\right|$ et $dy = \left|Y_f - Y_i\right|$.

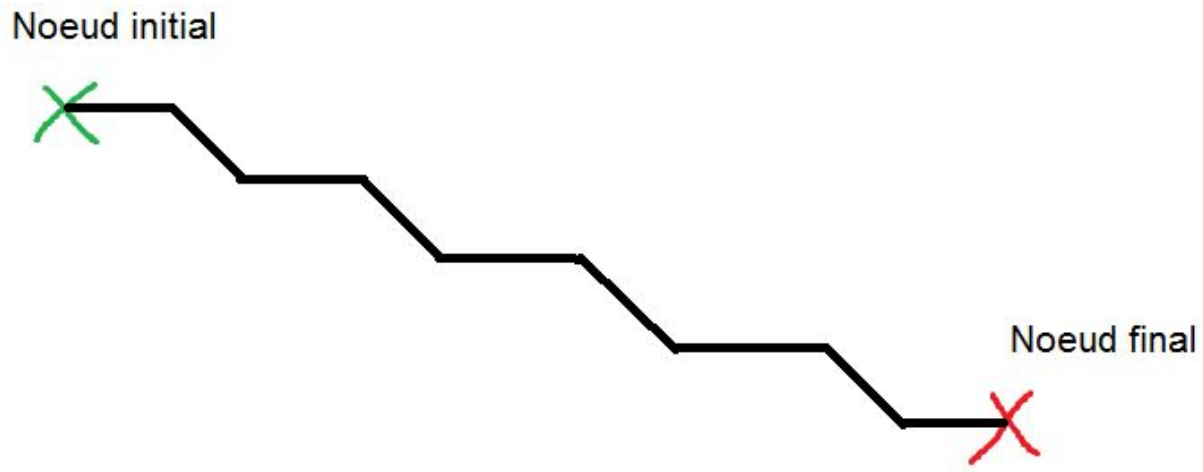

Figure 6 : Illustration de la distance Manhattan incluant des diagonales.

5 - Heuristique par découpage de tronçons

Cette dernière heuristique est liée à celle basée sur la distance euclidienne directe entre le noeud courant et le noeud final. Cependant, dans cette heuristique nous avons essayé de prendre en compte l'ensemble des variables qui influencent le temps de trajet du voilier, à savoir la direction/vitesse du vente, et la direction du voilier. Pour ce faire, nous avons découpé la distance euclidienne en tronçons de 10 kilomètres chacun. Nous calculons alors le temps de parcours de chacun de ces tronçons en appelant la fonction EstimationTemps(). Le temps total de trajet pour l'heuristique s'obtient en additionnant l'ensemble du temps de parcours de chacun des tronçons.Ainsi sont pris en compte dans le calcul de l'heuristique les différentes variables influençant le temps de trajet du voilier.

Tout ceci rend cette heuristique intéressante pour notre problème car elle concorde avec le fait que le voilier peut effectuer des trajectoires lisses, tout en prenant en compte différentes variables influençant le temps de trajet du voilier lors de son parcours réel.

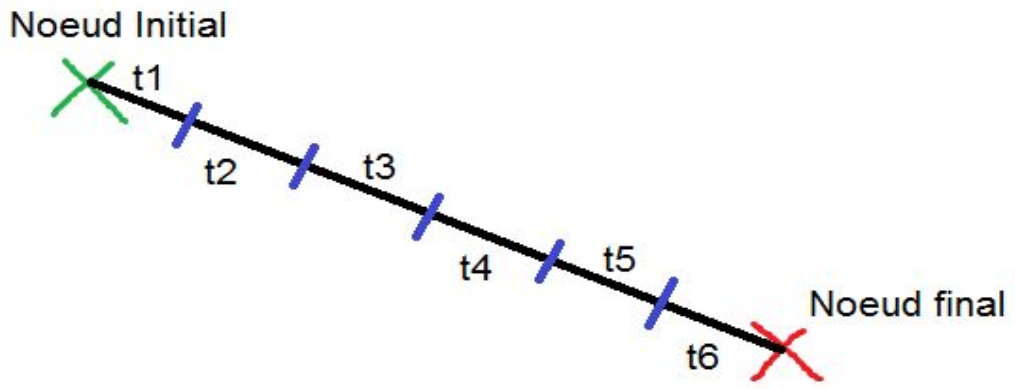

Figure 7 : Illustration de l'heuristique par tronçons. La distance totale d est découpée en n tronçons t_i de longueur inférieure ou égale à 10 kilomètres. Sur chaque tronçon, le temps de parcours du voilier est calculée en utilisant la fonction EstimationTemps().

IV - Résultats

Pour cette partie résultat nous avons utilisé comme heuristique celle par distance euclidienne (voir III.D.4) car elle permettait le meilleur compromis entre temps de calcul et temps de trajet "optimal" du voilier. Les autres heuristiques engendraient des temps de calcul bien plus longs et des résultats bien moins satisfaisants.
Nous avons de plus utilisé un pavage en carré 8 pour l'ensemble de nos résultats.

A - Cas a

Dans ce cas présent le voilier semble décrire une ligne quasiment droite entre le point initial et le point final.

- Temps de parcours : 3.689 heures
- Noeuds dans la solution : 30
- Noeuds Ouverts + Fermés : 2467
- Temps d'exécution : 2 secondes

Figure 8 : Résultat de l'algorithme A* pour le cas a en utilisant un pavage carré de 8 et l'heuristique par distance euclidienne

B - Cas b

Pour ce cas, le chemin emprunté par le voilier semble étrange, mais c'est cependant le meilleur temps de trajet obtenu pour ce cas donné. On remarquera que le temps d'exécution du programme pour trouver une solution à A* est extrêmement long comparé aux autres cas (3 min environ). Ce problème a été rencontré pendant toute la durée du projet peu importe l'heuristique utilisée. Pour certaines combinaisons de pavages/heuristiques, le temps de calcul de la solution s'élevait à plus de 25 minutes voire plus d'une heure sans jamais trouver de solution.

- Temps de parcours : 15.607 heures
- Noeuds dans la solution : 21
- Noeuds Ouverts + Fermés : 12999
- Temps d'exécution : 3 min 28 sec

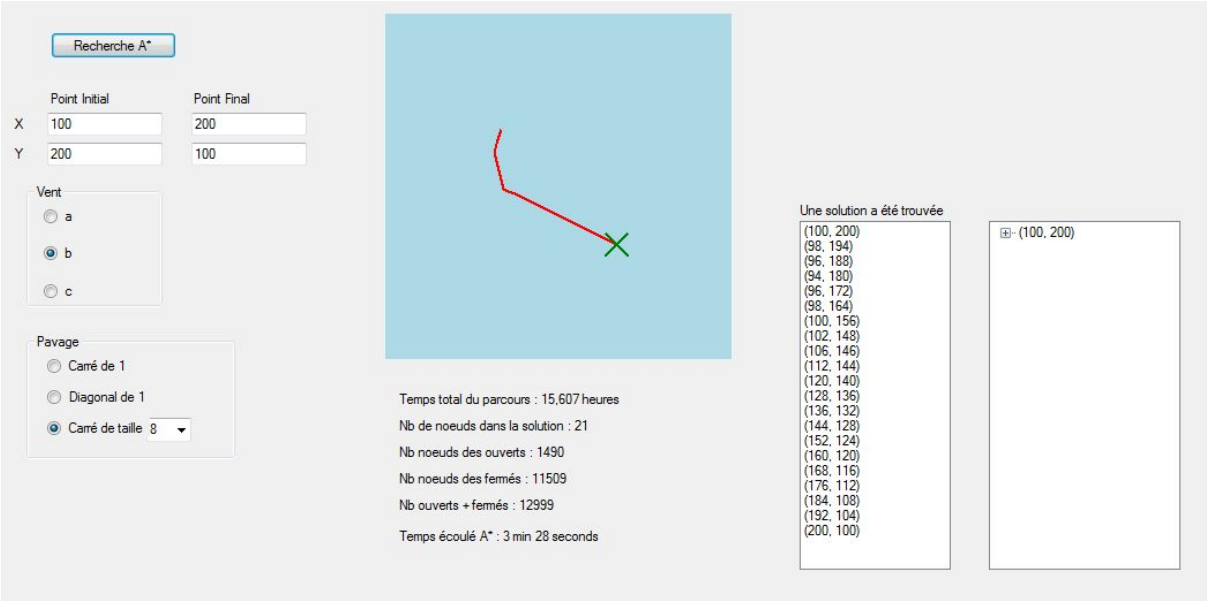

Figure 9 : Résultat de l'algorithme A* pour le cas b en utilisant un pavage carré de 8 et l'heuristique par distance euclidienne

C - Cas c

- Temps de parcours : 5.486 heures
- Noeuds dans la solution : 21
- Noeuds Ouverts + Fermés : 2737
- Temps d'exécution : 2 sec

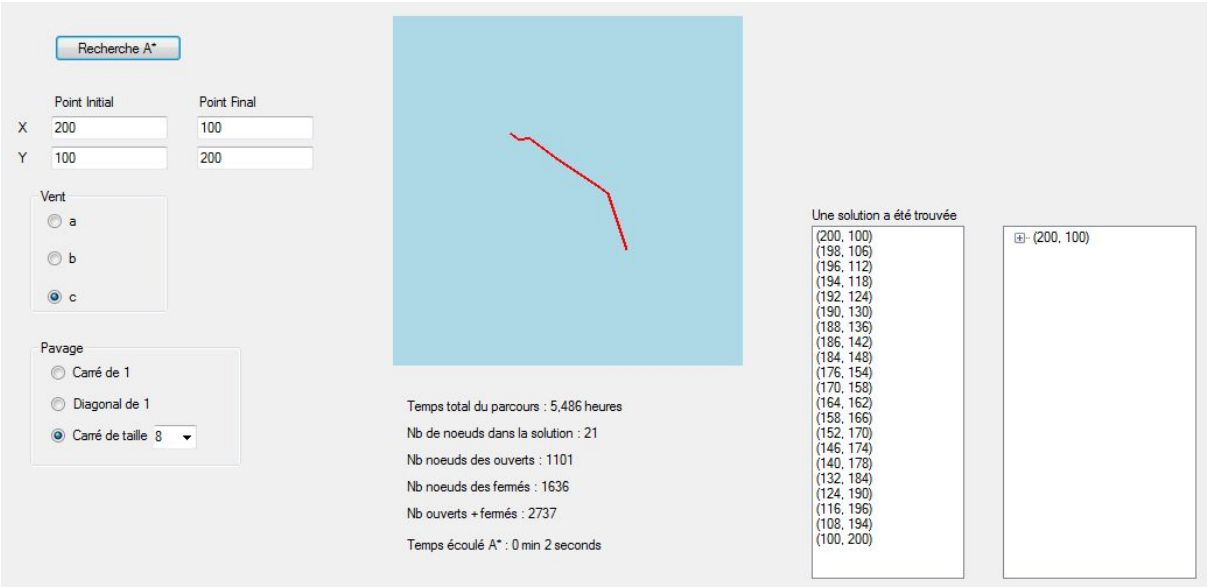

Figure 10 : Résultat de l'algorithme A* pour le cas c en utilisant un pavage carré de 8 et l'heuristique par distance euclidienne

Références

- http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html?fbclid=IwAR27vG8W8QzNtYTXA-zqx9TqzxTSzNu7L1WDw1QQ3-B-z7T0NNLoTeNLaoc
- Zhang, C. Li and W. Bi, "Rectangle expansion A* pathfinding for grid maps[J]", *Chinese Journal of Aeronautics (English Edition)*, vol. 29, no. 5, pp. 1385-1396, August 2016. https://doi.org/10.1016/j.cja.2016.04.023