École Nationale Supérieure de Cognitique

UE INGÉNIERIE COGNITIVE

INTELLIGENCE ARTIFICIELLE

Projet IA navigation à voile

Rapport écrit

Elève :

CARACATSANIS Oriane

PELLETREAU-DURIS Tom

VINCENT Amaury

Enseignant :

ADOLPHE Maxime

Novembre 2020

Bordeaux INP
ENSC

INTRODUCTION

Contexte

Ce projet a été réalisé dans le cadre du module d'intelligence artificielle à l'ENSC. L'objectif est d'aborder quelques notions répandues en intelligence artificielle et de les mettre en pratique. Ici nous nous familiarisons avec l'algorithme A*, un algorithme de recherche de meilleur chemin au sein d'un graphe. En particulier, il nous fallait réaliser un programme qui détermine le trajet optimal pour un voilier entre deux points.

Pré-requis / L'existant

Pour réaliser ce projet, il nous a été fourni en langage C# l'algorithme du plus court chemin ainsi qu'une classe GenericNode composée de méthodes abstraites qu'il nous a fallu surcharger dans nos classes.
L'algorithme A*, extension de l'algorithme de Dijkstra, nous a permis de déterminer le meilleur temps possible pour réaliser un déplacement. Pour cela, nous avons dû définir des heuristiques pertinentes pour répondre à notre problématique.

Objectif

Notre projet s'est divisé en deux grandes parties. En premier lieu, nous avons modélisé le problème, implémenté les heuristiques et créé notre interface utilisateur. Dans un deuxième temps, nous avons pu modifier, améliorer et valider ce programme grâce aux différents tests que nous avons effectués.

II.Modélisation du problème

II.1. Principes généraux

L'objectif de ce projet est de déterminer le trajet d'un voilier entre deux points de façon à ce que celui-ci soit parcouru en un temps minimal . Autrement dit, il faut que la valeur de la fonction time_estimation() pour un trajet donné parmis tous les trajets possibles entre les deux points en question, soit la plus petite possible et donc que la vitesse du bateau soit la plus grande possible. On distingue deux catégories possibles.

Pour chaque trajet entre le point de départ et le point d'arrivée, il existe un grand nombre de nœuds par lesquels le bateau passe. Nous cherchons le trajet avec le coût le plus faible. Pour cela, la stratégie d'exploration est fondée sur une modélisation d'un bateau. A partir d'un point, celui-ci peut se déplacer dans son voisinage proche. En fonction de la distance à parcourir, de la direction du bateau et de la direction du vent, la vitesse du bateau change.

La première possibilité est que la distance entre les deux points soit supérieure à 10 km. Dans ce cas, la fonction retourne une valeur d'un million pour signifier que ce déplacement est impossible.

La seconde possibilité est que la distance entre les deux points soit inférieure à 10 km. Ici, le temps est estimé en fonction de la direction et de la vitesse du vent. La direction du vent permet de déterminer la valeur de la variable α, qui est la direction du bateau par rapport à la direction du vent. La formule à appliquer diffère ici selon la valeur de α :

- Si $0 < \alpha < 45$: vitesse du bateau = (0.6 + (0.3α/45)*vitesse du vent.
- Si $45 < \alpha < 90$: vitesse du bateau = (0.9-((0.2(α-45))/45)*vitesse du vent.
- Si $90 < \alpha < 150$: vitesse du bateau = 0.7(1-((α90)/60)*vitesse du vent.
- Si $\alpha > 150$: vitesse du bateau = 0 puisque cela signifie qu'il est face au vent et qu'il ne peut pas avancer.

II.2. Algorithme A*

L'algorithme A*, découvert en 1968, est une amélioration de l'algorithme de Dijkstra lui-même découvert en 1959. Tout problème peut être résolu par la méthode A* s'il est résumable par un ensemble de nœuds dans un graphe. Ainsi, A*, pour fonctionner, nécessite la définition de nœuds, par exemple au sein d'une grille, de lien entre ces nœuds et d'une notion de coût (distance, temps, énergie nécessaire) pour passer d'un nœud à un autre. Alors l'algorithme pourra "raisonner" et par là trouver le chemin le moins coûteux entre deux nœuds s'il sont liés entre eux par d'autres nœuds. Il est caractérisé

par l'intervention, dans le raisonnement, d'une "heuristique". L'heuristique est une information permettant d'orienter le chemin vers une destination finale. L'heuristique, il faut bien le comprendre, est une donnée propre à chaque nœud permettant de le discriminer d'autres nœuds moins efficaces d'un point de vue global. Une manière simple de se représenter cela est d'imaginer une carte quadrillée ou chaque case est un nœud comme peut le voir sur le schéma ci-dessous où le Vert est le nœud initial, Rouge le nœud final, Bleu des nœuds impossibles. Sur chaque case on peut observer en bas à gauche le coût réel, en bas à droite le coût heuristique et en haut à gauche le coût total. L'heuristique peut par exemple être la distance en "vol d'oiseaux" du point par rapport à l'autre ce qui mathématiquement revient à calculer AB = $\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

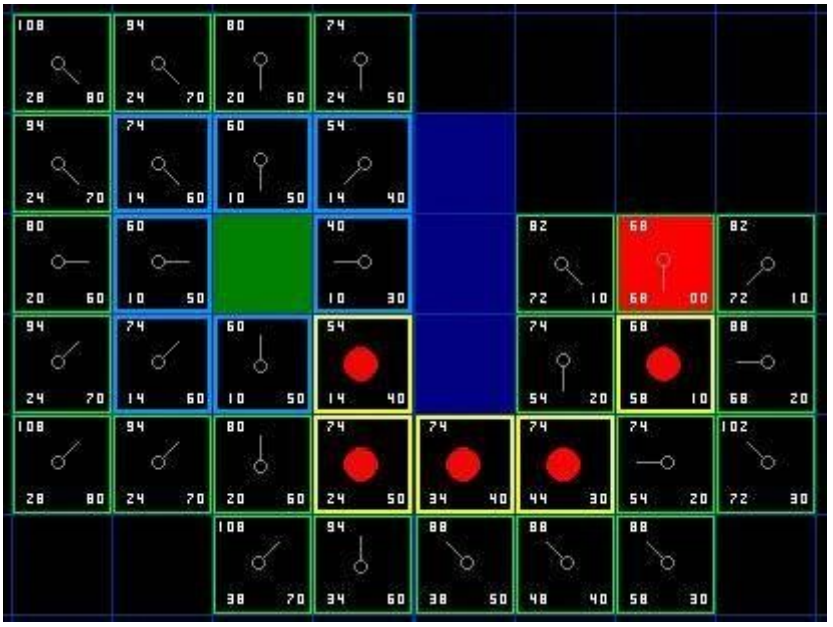


Schéma représentatif de l'algorithme A appliqué à un problème de chemin dans une grille.*

L'algorithme A*, comme nous l'avons vu, est un algorithme qui vise à trouver le meilleur chemin par une méthode de recherche optimisée. Dans le cas de notre bateau, il s'agit d'adapter les notions de coût et d'heuristique. En effet, ici le coût de déplacement est inversement proportionnel à la vitesse du bateau, plus un bateau va lentement entre deux points plus ce déplacement est coûteux. On peut aussi utiliser une notion de temps qui elle est proportionnelle. Plus le bateau met longtemps à se déplacer, plus le coût est élevé. Pour l'heuristique, la notion de distance à "vol d'oiseau" ne suffit plus car le coût est exprimé en temps (seconde par exemple). De cette manière notre heuristique est plutôt le temps minimum de voyage entre deux points.

Dans cette logique, nous avons modélisé le problème à partir de plusieurs classe : une classe Node.cs héritée de GenericNode.cs qui permet d'initialiser chaque noeud et qui inclut les fonctions de coût général (timeEstimation()) et d'heuristique(). Une classe SearchTree.cs qui contient les

fonctions de recherches de l'algorithme A* que sont ChercheNodeDansFermes(), ChercheNodeDansOuvert, RechercheSOlutionAEtoile(), MAJSuccesseurs(), InsertNewModeInOpenList, GetSearchTree() et AjouteBranche(). Enfin nous avons implémenté une interface Form1.cs qui sera détaillée plus bas.

C'est particulièrement la fonction Node() qui nous est important de détailler puisque nous l'avons codé entièrement. Les principales fonctions de cette classe sont les suivantes :

Nom	Entrée(s)	Sortie(s)	Description
Node	double px, double py, char c		Constructeur
GetListSucc		List<GenericNode>	Renvoie une liste de successeurs pour un état donné du voilier. Correspond à la définition du voisinage.
CalculeHCost		double Hcost	Renvoie l'heuristique.
GetArcCost		double temps	Calcule le coût réel G entre la case initiale et son successeur. Cette fonction appelle time_estimation.
IsEqual		Bool equal	Compare deux nœuds et renvoie true si le nœud en paramètre est le même que le nœud actuel.
EndState		bool	Renvoie true si la case où se situe le voilier est la case finale.
time_estimation	x1,y1 ; x2,y2 Coordonnées de deux points	double distance / baotspeed	Calcule le temps de déplacement pour le bateau entre deux point en fonction du vent
Heuristique	x1,y1 ; x2,y2 Coordonnées de deux points	double distance / baotspeed	Calcule l'heuristique entre deux points
get_wind_speed	x,y coordonnées d'un point	double wind_speed	
get_wind_directi	x,y	double	

on	coordonnées d'un point	wind_direction	

Tableau des méthodes dans la classe Node.cs

II.3. Choix des successeurs

Sachant qu'à partir d'un point, en noir ici, son voisinnage maximale est de 10 km et que notre pavage est en kilomètres unitaires, nous pouvons représenter le voisinnage théorique maximale comme une matrice de 21x21 où le centre représente le point auquel on s'intéresse et le cercle intérieure représente la distance maximale de 10 km. En effet, la distance en diagonale n'est pas de 10 km pour 10 cases en diagonales mais de sqrt(200). Nous avons alors pu considérer plusieurs possibilités de voisinage.

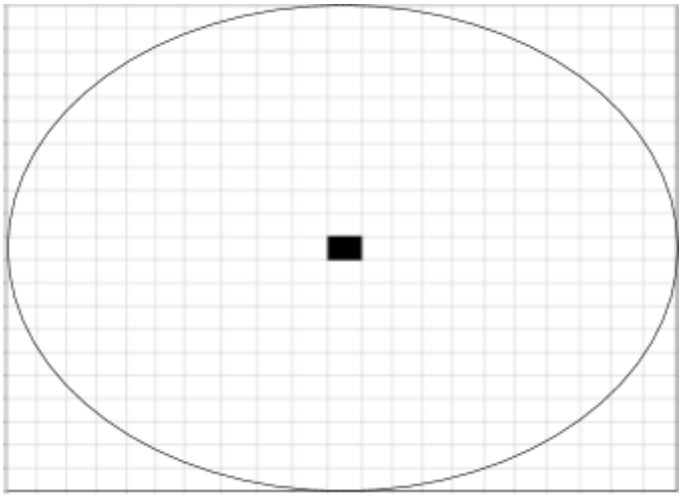


Schéma du voisinage nu

Le premier cas que nous avons étudié était une stratégie d'exploration pour un voisinage carré. Cette stratégie, simpliste, est chronophage car la propagation est particulièrement lente.

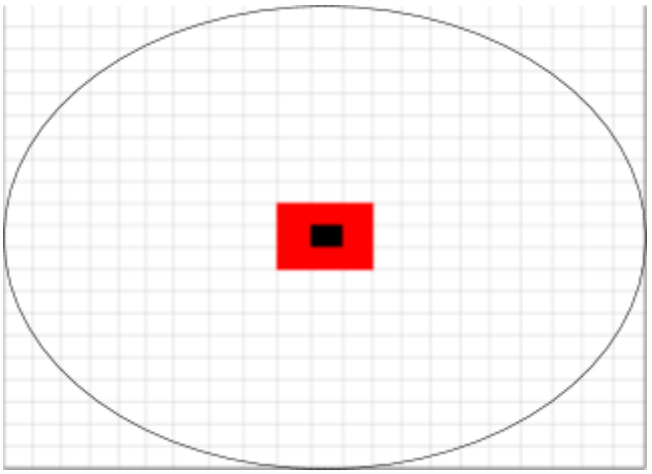


Schéma du voisinage carré simpliste

Une autre possibilité que nous avons envisagée était d'augmenter la portée de notre voisinage pour augmenter la taille de la liste ouverte et donc la rapidité de la propagation. Cela permettait ainsi de gagner du temps mais ce n'était pas très optimisé. En effet, cette méthode mettait toujours longtemps à charger. Cette méthode est illustrée pour les cas A, B et C.

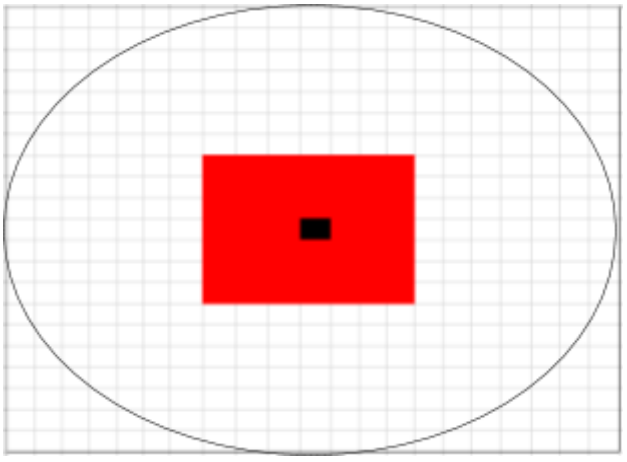


Schéma du voisinage carré avec une portée plus importante

Enfin, la solution alternative que nous avons choisi a été de pratiquer une exploration par niveau. Un voisinage central garanti d'accéder à notre destination finale puisque n'importe quel point de la grille est accessible en escalier (exploration via l'escalier de manhattan). L'exploration de points plus éloignés permet de gagner du temps à un certain niveau de l'exploration et de remplacer ainsi certains nœuds dans la liste ouverte.

Nous avons dans un premier temps essayer avec ce motif qui, bien qu'il permettait de trouver de bons résultats, était tout de même lent.

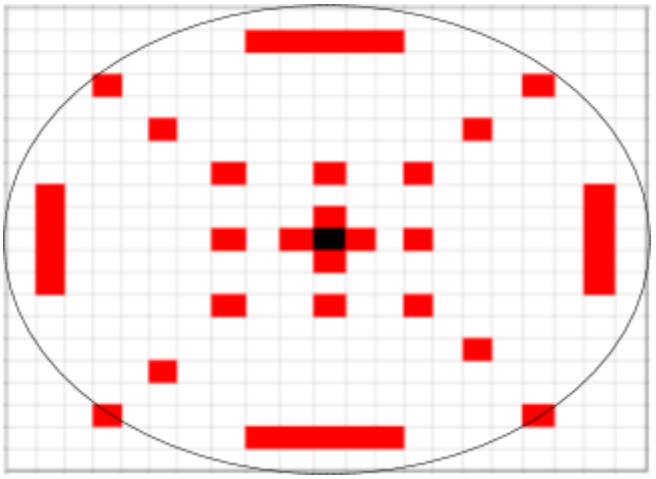


Schéma du voisinage en motif "araignée" que l'on pensait efficient

La meilleure solution que nous ayons trouvé est d'effectuer un pavage régulier sur la base de deux variables, une variable de zonage correspondant à un coefficient multiplicateur de distance et une variable de sélection qui met de côté un nœud sur n.
Par exemple, Avec une distance de 3 et un noeud sur trois on obtient le pavage suivant :

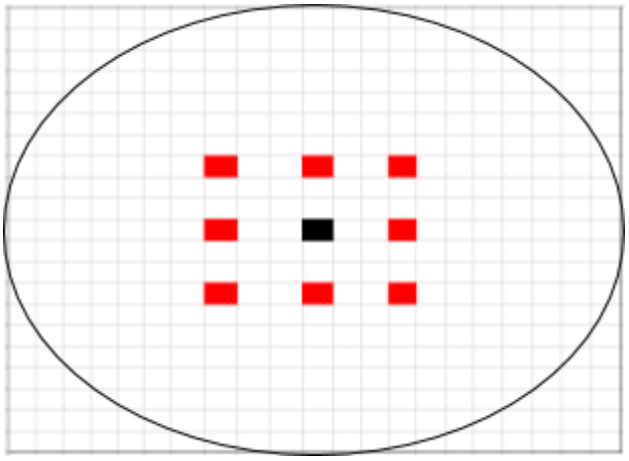


Schéma du voisinage avec une portée de 3 mais on ne prend qu'une ligne sur trois et qu'une colonne sur trois

Nous avons trouvé un bon compromis entre rapidité de calcul et exactitude des résultats en utilisant le pavage ci-dessous, sélectionné par défaut dans notre Form. On peut voir qu'il y a un pavage 0,5 puisque l'on prend un noeud tous les 2,5 sur un voisinage de 5 (2,5*2).

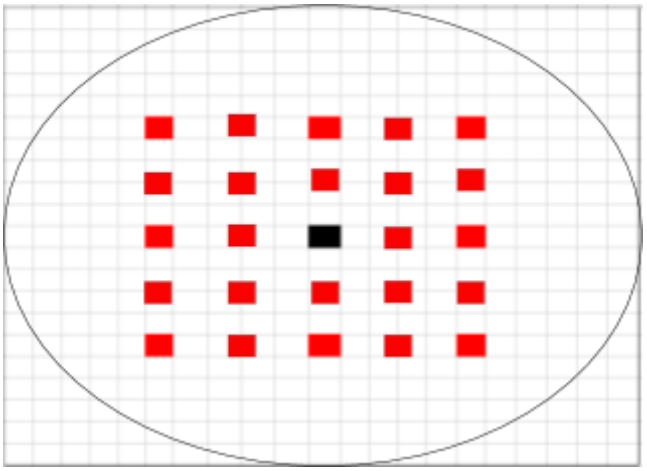


Schéma du voisinage avec une portée de 5 mais on ne prend qu'une ligne sur 2,5 et qu'une colonne sur 2,5

Nous avons aussi utiliser le pavage suivant, un sur deux avec un voisinage de 10 (2*5) comme on peut voir dans les résultats aux cas A, B et C.

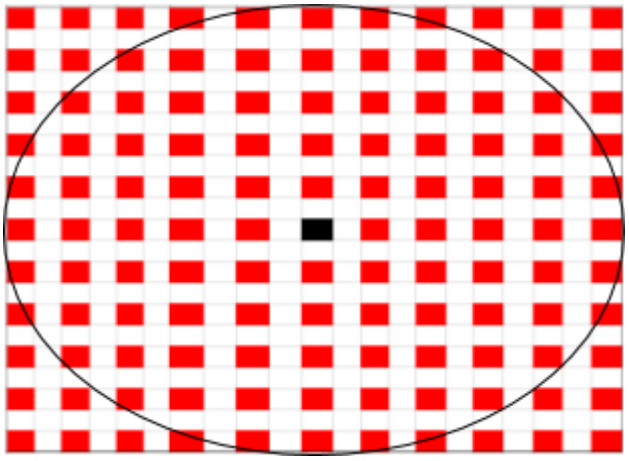


Schéma du voisinage avec une portée de 10 mais on ne prend qu'une ligne sur 2 et qu'une colonne sur 2

Finalement, le code de cette fonction est disponible en annexe.

II.4. Coût réel

Le coût réel ou coût général d'un déplacement du bateau entre deux points est relatif au temps. En effet, comme nous l'avons vu plus haut, nous aurions pu considérer que le coût était une fonction inversement proportionnelle de la vitesse de déplacement du bateau. Le temps de déplacement est donc parfaitement adéquat et correspond bien à un coût. Plus le temps de déplacement est grand, plus il est coûteux de se déplacer.

La fonction time_estimation() qui prend en entrée les coordonnées x,y de deux points se fonde donc sur le calcul d'une distance. Si la distance est supérieure à 10 km alors la fonction renvoie un million. Sinon, elle calcule la

direction et la vitesse du vent en fonction de la position du bateau (on prend le point à mi-chemin entre nos deux points de déplacements comme référence). On calcule ensuite la direction du bateau, ce qui correspond à l'angle entre notre vecteur de déplacement et la droite des abscisses. Puis on calcul la différence entre cet angle et l'angle de la direction du vent. En fonction de cet angle que l'on nomme alpha, nous sommes capable de dire quelle sera la vitesse de notre bateau. Cela correspond bien à la réalité.

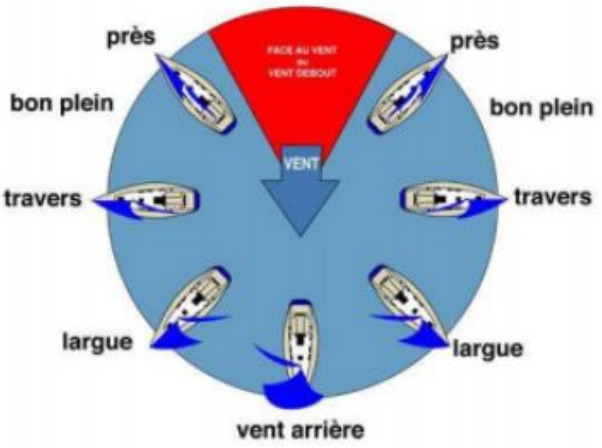


Schéma de la prise d'une voile en fonction de la direction du vent pour un bateau.

II.5. Heuristique

L'objectif de l'heuristique est de calculer le temps minimal de déplacement du bateau entre deux points, sans prendre en compte les contraintes extérieures, par exemple la direction du vent, sa vitesse ou la direction du bateau. On pourrait dire que c'est le temps à "vol d'oiseau" quelque part. Notre heuristique est toujours plus faible que le coût réel de déplacement du bateau.

Pour ce faire, nous avons choisi de fixer la valeur de α à 45. En effet, selon la figure X suivante, pour un vent de 50km/h, la vitesse du bateau est la plus élevée lorsque la valeur de α est égale à 45. Or l'objectif voulu ici est de calculer le temps optimal, soit le plus petit, donc d'obtenir la vitesse de bateau la plus grande.

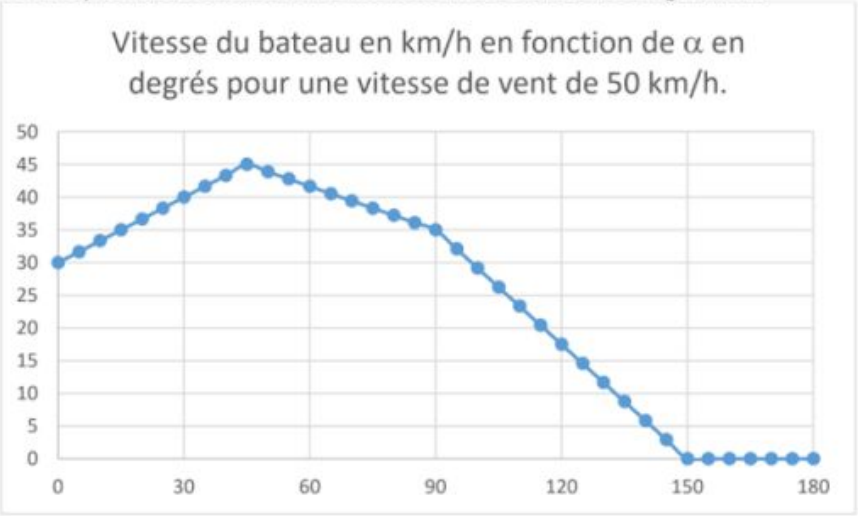


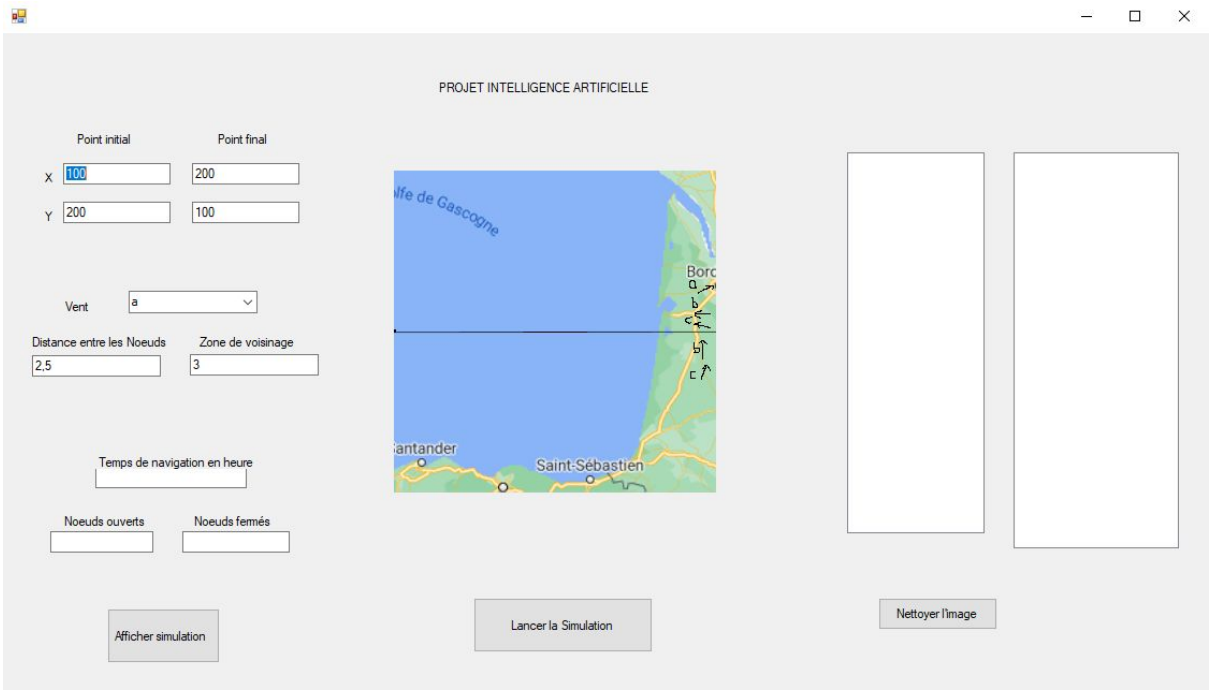
Schéma de la vitesse du bateau en fonction de la direction du vent.

La fonction heuristique, en plus de fixer α, doit aussi fixer la vitesse du vent. Ainsi, pour le cas "a", elle est de 50 km/h partout. Pour le cas "b" et "c", elle est de 50km/h pour la moitié supérieure de la carte et de 20km/h pour la moitié inférieure de la carte. On applique ensuite la formule correspondante afin de calculer la vitesse maximale du bateau, soit : vitesse du bateau = (0.6 + (0.3α/45)*vitesse du vent. Le tout sera divisé par la distance entre les points pour correspondre à une valeur temporelle.

La fonction CalculeHCost(), qui était à surcharger et qui permet donc de calculer l'heuristique dans GenericNode, utilise ensuite cette fonction pour renvoyer le coût heuristique.

III. Interface

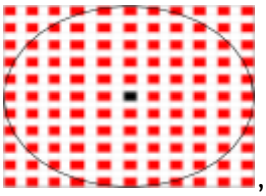
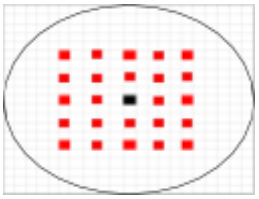
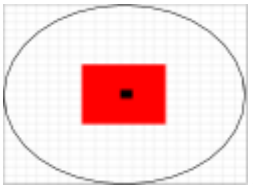
Notre interface est composée d'un seul form, qui s'affiche lorsque l'utilisateur lance le programme. Il est divisé en quatre parties.



Vu de notre Form avec une petite carte du Golf de Gascogne.

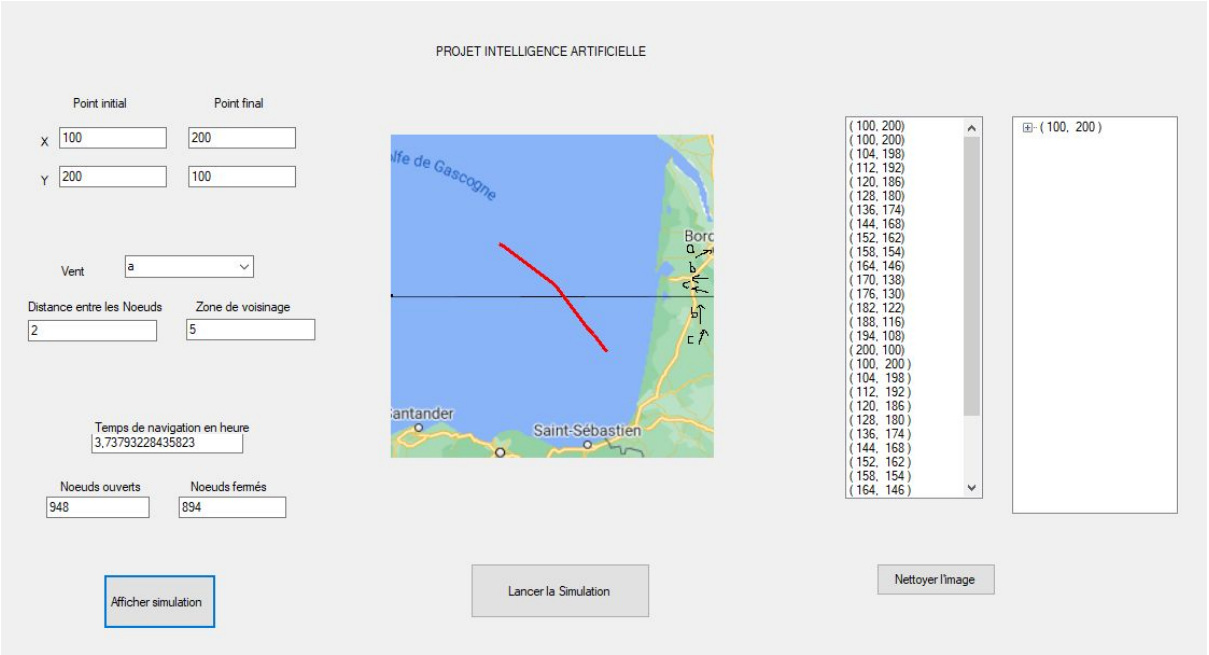
Dans la première partie, à gauche du form, l'utilisateur peut sélectionner les différents paramètres souhaités. C'est ici qu'il renseigne les coordonnées des points de départ et d'arrivée, le cas de figure "a", "b" ou "c"(ce qui modifie la direction et la vitesse du vent en fonction de la position du bateau), et enfin la distance entre deux nœuds et la zone de voisinage qui sont les paramètres changeant le voisinage comme expliqué plus haut.
La seconde partie, en bas à gauche, affiche les résultats de la simulation. On y trouve le temps total du meilleur trajet trouvé estimé par le programme, le nombre de nœuds ouverts et le nombre de nœuds fermés. Un bouton "Afficher simulation" est également présent. Ce bouton impacte directement la troisième partie du form puisqu'il permet de dessiner le trajet en rouge sur l'image présente au milieu du form. Sous cette image on trouve également un bouton "Lancer simulation" sur lequel l'utilisateur clique après avoir renseigné ses paramètres afin de lancer les calculs.
Enfin, la quatrième et dernière partie se situe à droite du form. La totalité des noeuds par lesquels on passe lors de notre parcours s'affiche ici. On a enfin un bouton "Nettoyer image" qui permet à l'utilisateur de réinitialiser le form et de relancer une simulation avec des paramètres différents s'il le souhaite.

IV. Résultats obtenus

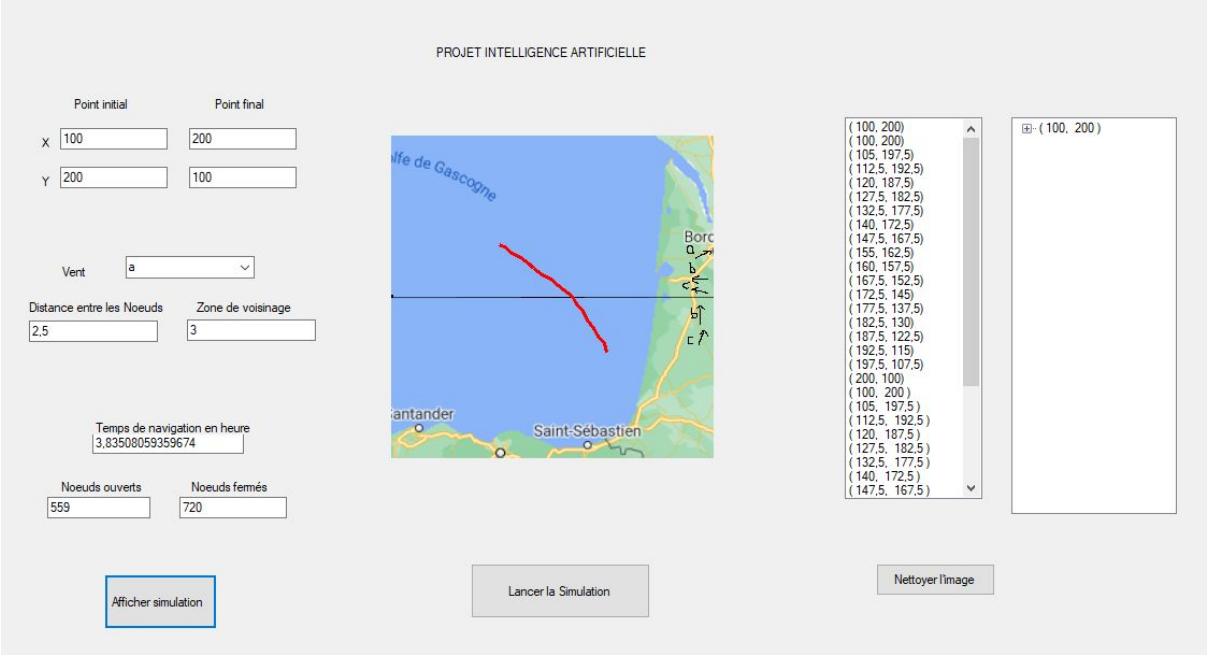
Pour chaque cas nous avons montré ce que nous obtenions avec le pavage (2,5)  ,
le pavage (2,5,3) 
et le pavage (1,3)  .
On a pu voir que le premier est le plus rapide en termes de temps de calcul, que le dernier est le plus précis et donc permet de trouver le trajet le plus rapide. Le deuxième est donc notre compromis entre temps de calcul et finesse de trajet.

IV.1. Cas A

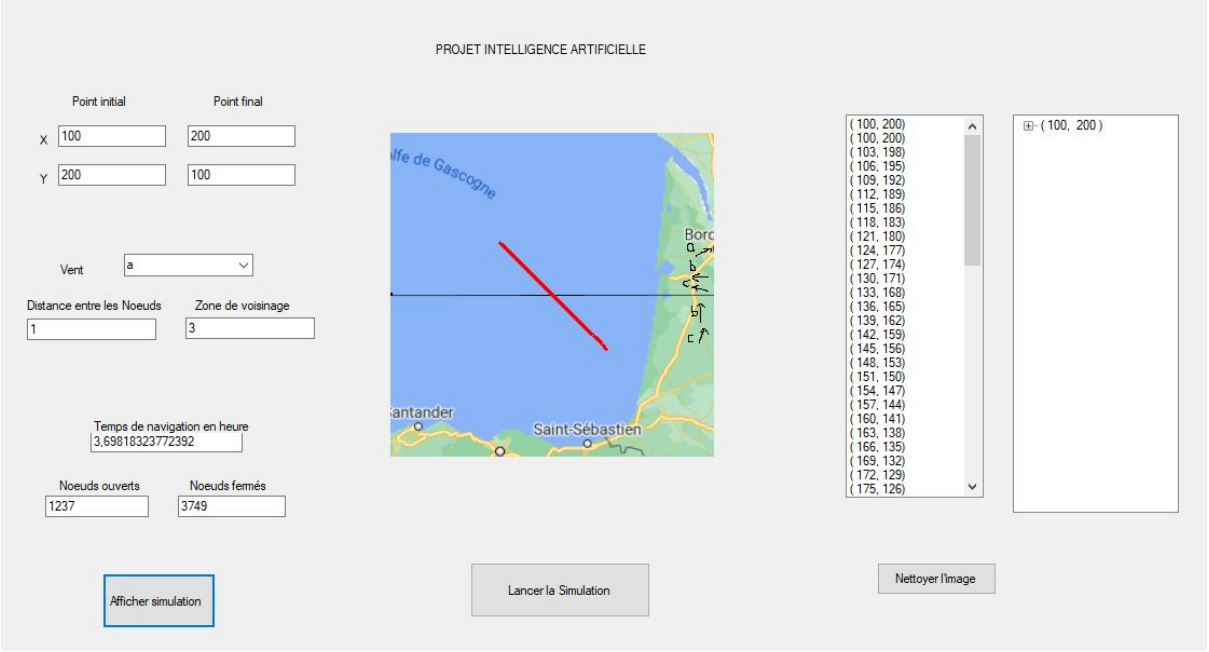
Pour le cas A nous avons trouvé un trajet optimal de durée 3,69 heures. La trajectoire est presque oblique ce qui est très bien puisque le vent va vers le Nord-Est. Le bateau se comporte comme il le ferait avec une voile et du vent.



Vue du form pour le cas A avec les paramètres (2;5) pour le voisinage.



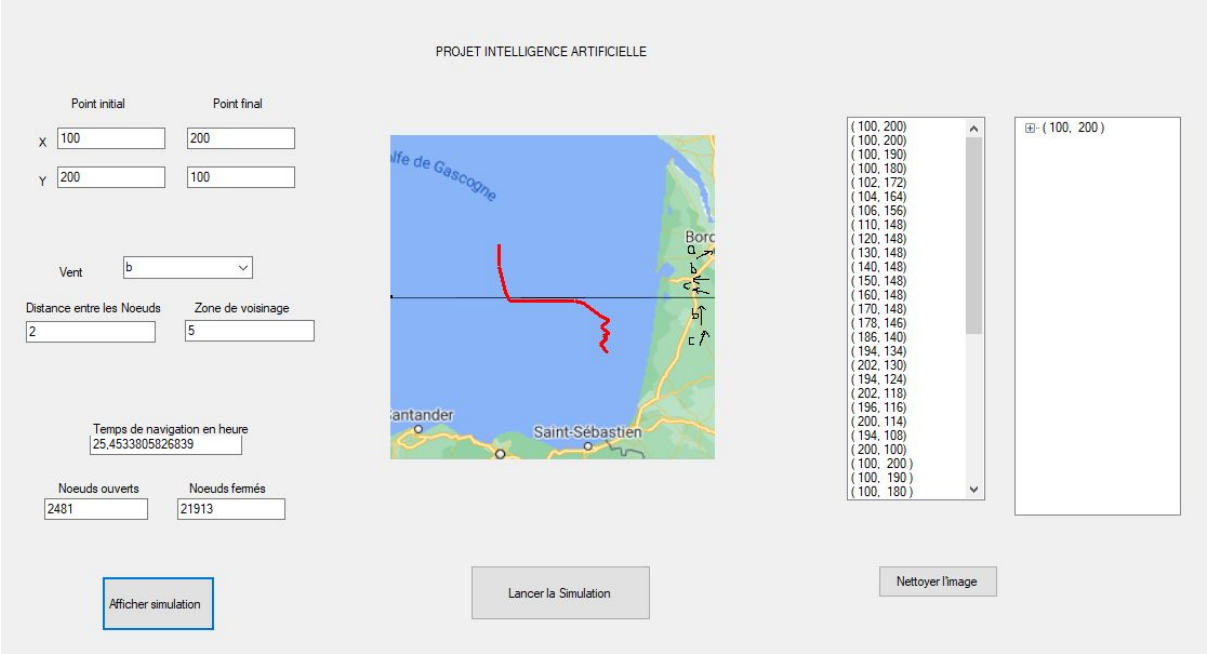
Vue du form pour le cas A avec les paramètres (2,5 ; 3) pour le voisinage.



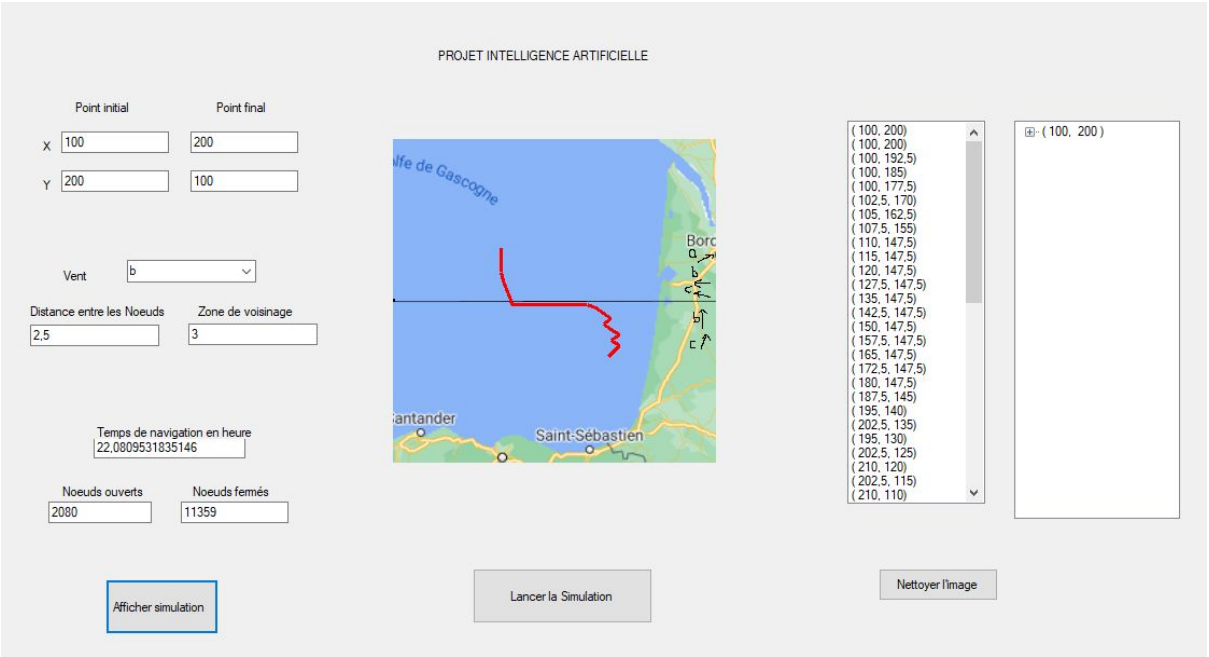
Vue du form pour le cas A avec les paramètres (1;3) pour le voisinage.

IV.2. Cas B

Pour le cas **B** nous avons trouvé un trajet optimal de durée 22,08 heures. La trajectoire est curieuse mais correspond bien puisque le vent va vers l'ouest dans la première moitié et va vers le Nord dans la moitié inférieure. Le bateau se comporte comme il le ferait avec une voile et du vent : ce que l'on peut constater grâce au zig-zag à la fin.



Vue du form pour le cas B avec les paramètres (2,5) pour le voisinage.

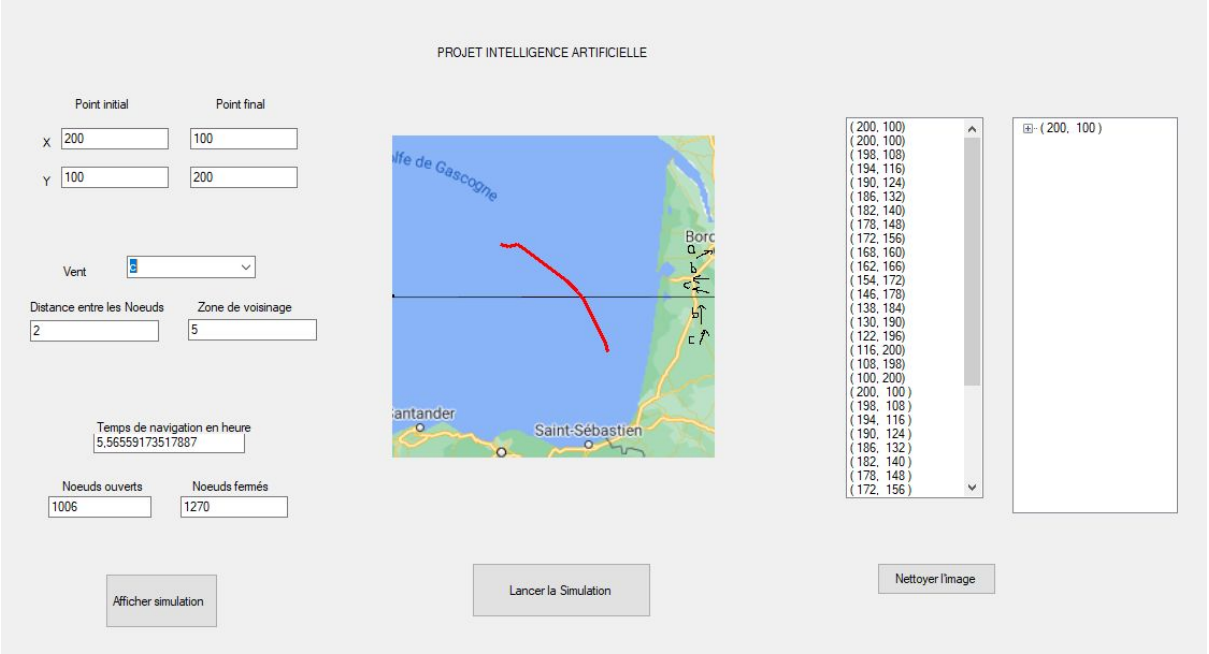


Vue du form pour le cas B avec les paramètres (2,5 ; 3) pour le voisinage.

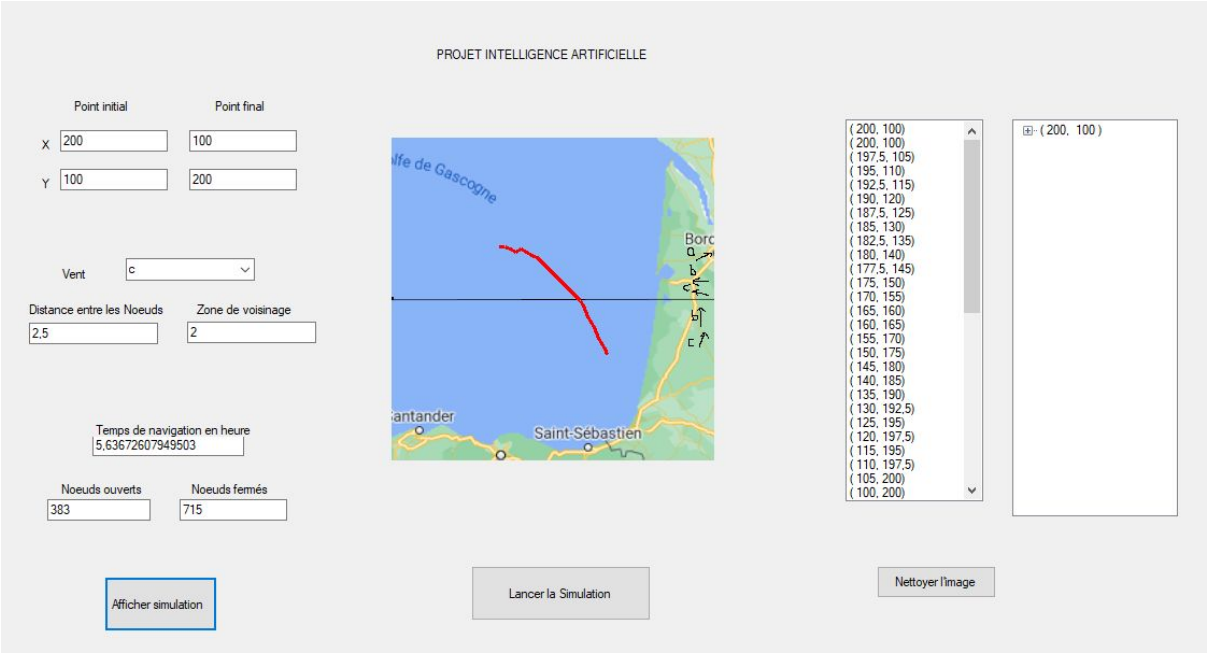
IV.3. Cas C

Pour le cas **C** nous avons trouvé un trajet optimal de durée 5,56 heures. La trajectoire est bonne puisque le vent va vers l'Ouest-Nord-Ouest dans la première moitié et va vers le Nord-Nord-Est dans la moitié inférieure. Le

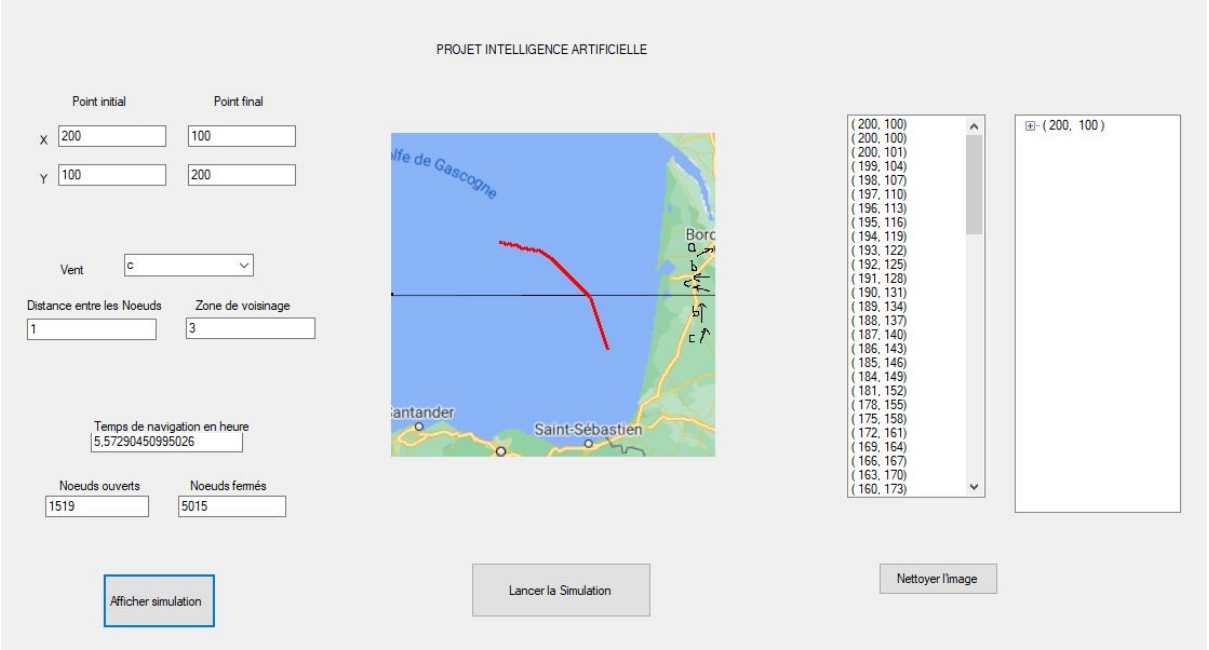
bateau se comporte comme il le ferait avec une voile et du vent : ce que l'on peut constater grâce au zig-zag à la fin.



Vue du form pour le cas C avec les paramètres (2,5) pour le voisinage.



Vue du form pour le cas C avec les paramètres (2,5 ; 3) pour le voisinage.



Vue du form pour le cas C avec les paramètres (1 ; 3) pour le voisinage.

Conclusion

En conclusion, ce qui est le plus marquant pour nous c'est la sensation que l'algorithme ait un comportement proche de la réalité sans que nous l'ayons codé comme tel. En effet, comme le montre la trajectoire, le bateau prend souvent le vent en largue pour aller le plus vite possible, il change de bord lorsque c'est nécessaire, et fait même des zig-zags en navigation au près pour éviter d'être vent debout. Tout cela correspond évidemment à la navigation humaine, ce qui prouve que l'être humain à un comportement optimal de part son expérience et que notre algorithme fonctionne bien.
De plus, ce projet nous a permis d'approcher l'algorithme A* d'une manière plus palpable. Les nombreux essais, les changements d'heuristique et de voisinage nous ont permis de mieux l'appréhender.
Enfin, le travail de recherche nous a permit de comparer l'algorithme A* à ses prédécesseurs, ce qui nous sera peut être utile dans d'autres applications.

Bibliographie

Exemple littérale :

https://web.archive.org/web/20170505034417/http://blog.two-cats.com/2014/06/a-star-example/
Explication littérale :
https://web.archive.org/web/20170509000025/http://www.policyalmanac.org/games/aStarTutorial.htm
Exemple de visualisation :
https://www.youtube.com/watch?v=19h1g22hby8
Explication par la chaîne numberphile :
https://www.youtube.com/watch?v=ySN5Wnu88nE