Domecq François
Laran Baptiste
Leger Corentin

Bordeaux INP
ENSC

Rapport Projet Intelligence Artificielle

Ce document intègre des fichiers GIF. Cependant, ces fichiers ne sont pas visionnables sur un document PDF. C'est pourquoi nous vous mettons à disposition le lien vers notre Google Docs pour que vous puissiez lire, si vous le souhaitez, la version de notre document avec les vidéos fonctionnelles. Veuillez nous excuser pour la gêne occasionnée.

Domecq_Laran_Leger_Rapport_IA

2ème Année (2021-2022)

Introduction

L'objectif de ce projet est d'exploiter l'algorithme A* et de trouver des heuristiques pour réduire la taille de l'arbre d'exploration. Il nous a été fourni un programme C# où l'algorithme A* est implémenté mais sans heuristique ce qui revient à appliquer l'algorithme de Dijkstra.

Le programme C# implémente 3 environnements différents présentés sur la figure ci-dessous :

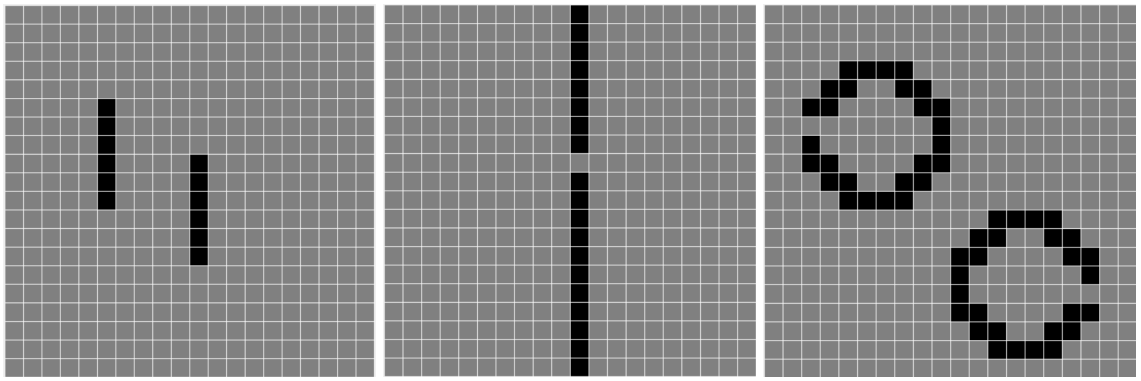


figure 1 : Environnement 1, 2 et 3

Il nous est donc demandé de modifier la fonction CalculeHCost du programme, qui pour le moment renvoie 0 comme valeur, en fonction de chaque environnement afin d'appliquer l'heuristique le plus adapté à chaque situation.

I - Gestion du projet

Répartition des tâches

Tâche	Effectif
Compréhension du code fourni	François, Baptiste & Corentin
Programme pour générer les listes de nombre	François
Environnement 1	Baptiste & Corentin
Environnement 2	François, Baptiste & Corentin
Environnement 3	François, Baptiste & Corentin
Rapport	François, Baptiste & Corentin

Pour la répartition des tâches, nous avons commencé par tous lire le code qui nous était fourni pour en comprendre l'ensemble. Dans un second temps, François s'est occupé de la partie qui allait nous permettre d'effectuer nos tests pendant que Baptiste et Corentin trouvaient l'heuristique le plus adapté à l'environnement 1. Ces deux tâches ayant été terminées en même temps, nous avons décidé de collaborer sur les deux environnements restants ainsi que pour la rédaction du rapport

II - Résultats

Méthode de test

Dans un premier temps, pour réaliser nos tests, nous avons modifié manuellement les coordonnées des noeuds de départ et d'arrivée pour couvrir les différents cas possibles pour chaque environnement. Puis dans un second temps, nous avons créé un programme (cf. Annexe) nous permettant de générer une liste de 1000 nombres aléatoires. Nous avons ensuite défini quatres tableaux de 1000 nombres : xInitial, yInitial, xFinal et yFinal. Nous utilisons ensuite ces tableaux dans la classe Form1.cs. Lorsque l'utilisateur clique sur le bouton A* dans le Form, une boucle for avec i allant de 0 à 999 se déclenche et le programme trouve un chemin pour chaque paire :

(xInitial[i], yInitial[i]) -> (xFinal[i], yFinal[i])

Nous avons également pris soin d'éviter de générer des coordonnées placées sur des obstacles pour s'assurer qu'un chemin existe. Nous avons donc 4 listes de coordonnées pour chaque environnement. Une fois la boucle for parcourue, nous calculons la moyenne du nombre de nœuds parcourus sur les 1000 itérations. Nous pouvons ainsi comparer l'efficacité de nos heuristiques par rapport à l'algorithme de Dijkstra sur un nombre d'essais bien plus grand que lors d'une comparaison un à un.

Nos résultats sont donc présentés et illustrés à l'aide de vidéos dans la section ci-dessous. Cependant, les vidéos affichent des cas où le chemin est jaune et non bleu. Cela est dû à un "bug" d'affichage dû aux 1000 itérations. En effet, nous avons pris soin de vérifier qu'un chemin était bien trouvé à chaque itération. Enfin, nous présentons également un tableau comparatif de la réussite de notre algorithme lorsque nous modifions les coordonnées manuellement.

Environnement 1

Le code utilisé pour l'environnement 1 est disponible en annexe. Pour l'environnement 1, nous avons souhaité tester plusieurs heuristiques afin de comparer leur efficacité.

a) Algorithme de Dijkstra

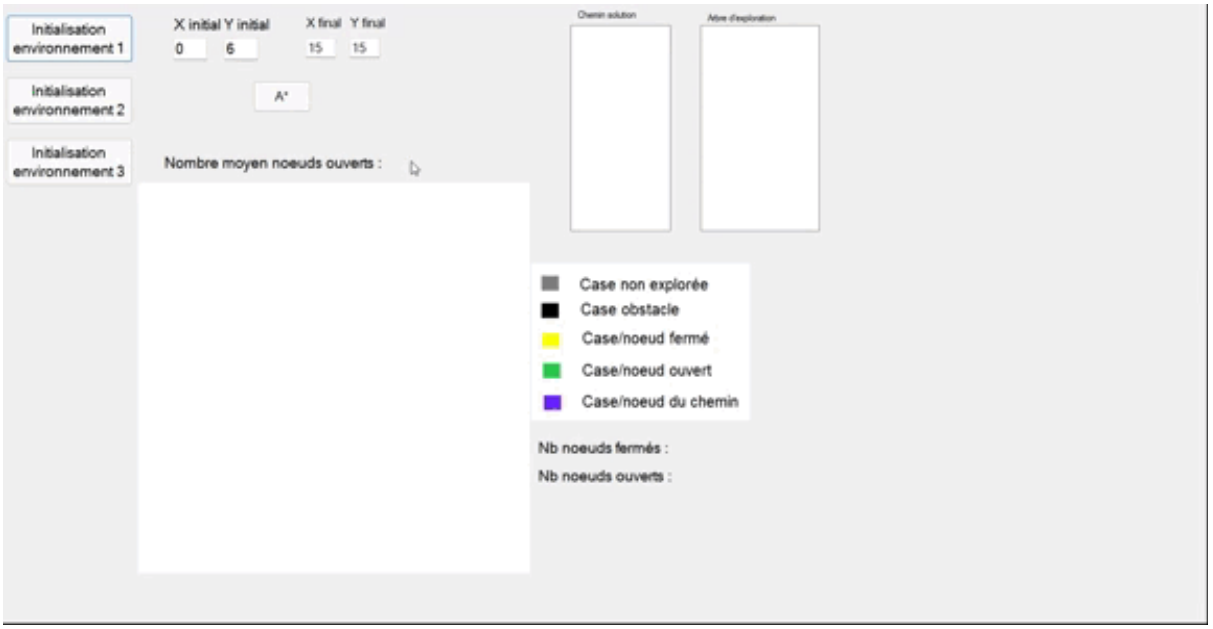


figure 2 : Vidéo de l'algorithme de Dijkstra

$$H(n) = 0$$

Dans le cas de l'algorithme de Dijkstra, c'est-à-dire où l'heuristique est nul. Nous obtenons une **moyenne de 218 nœuds ouverts**.

b) Distance de Manhattan

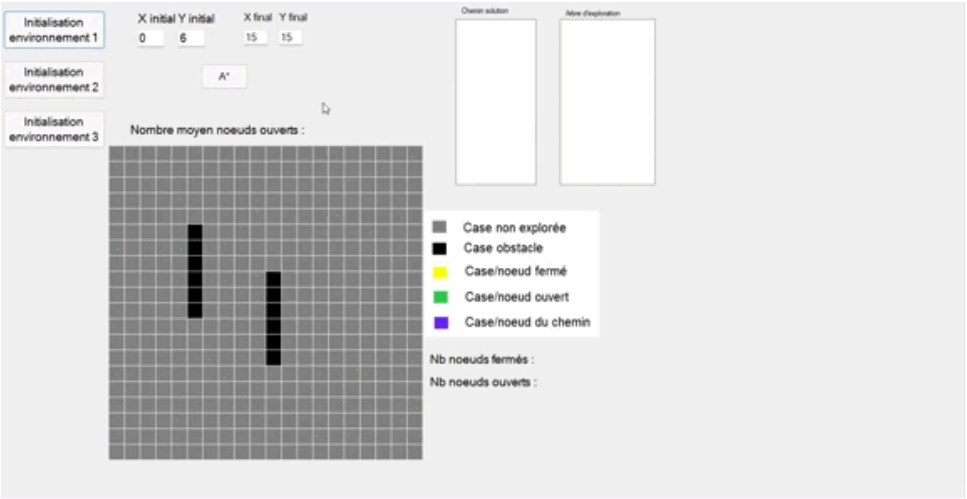


figure 3 : Vidéo de l'algorithme A* avec la distance de Manhattan comme heuristique

$$H(n) = |xf - x| + |yf - y|$$

Dans un premier temps, nous avons essayé d'appliquer l'heuristique Distance de Manhattan. C'est un heuristique standard dans le cas des grilles carrées. Cet algorithme s'est révélé être très efficace avec une **moyenne de 40 nœuds ouverts** sur 1000 essais. Cependant, cette heuristique ne prend pas en compte la possibilité d'effectuer des mouvements en diagonale. Nous avons décidé d'essayer d'autres heuristiques prenant en compte cette possibilité.

c) Distance de chebyshev

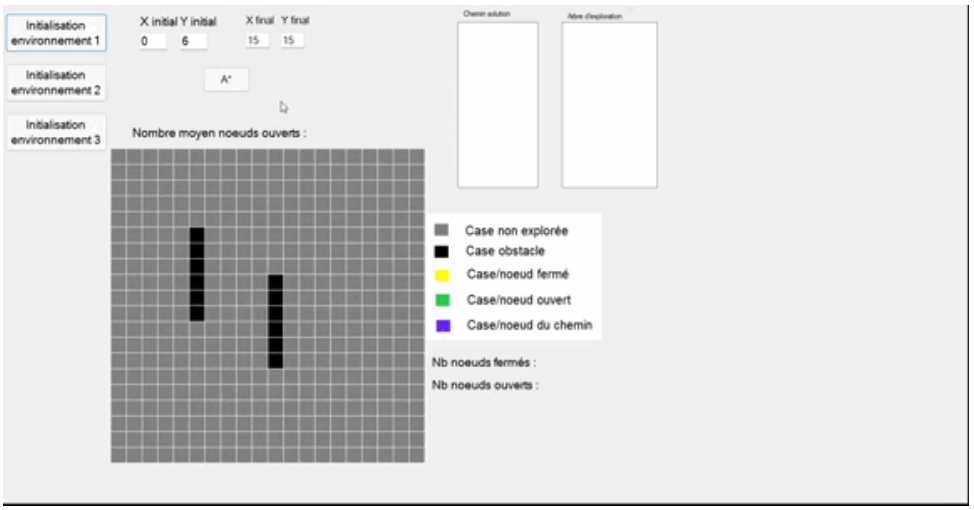


figure 3 : Vidéo de l'algorithme A* avec la distance de Chebyshev comme heuristique

$$H(n) = |xf - x| + |yf - y| - min(|xf - x|, |yf - y|)$$

Étonnamment, lorsque nous appliquons la distance de Chebyshev (qui prend en compte la possibilité d'effectuer des déplacements en diagonale), nous obtenons une **moyenne de 74 nœuds ouverts**. Soit une moyenne supérieure à celle où nous utilisons la distance de Manhattan.

d) Distance Euclidienne

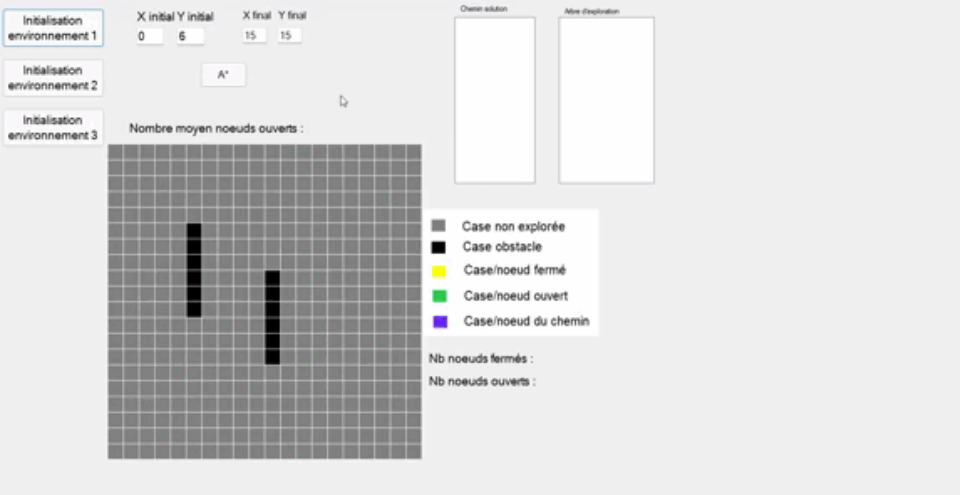


figure 4 : Vidéo de l'algorithme A* avec la distance euclidienne comme heuristique

$$H(n) = \sqrt{(xf - x)^2 + (yf - y)^2}$$

L'heuristique Distance Euclidienne prend également en compte la possibilité d'effectuer des déplacements en diagonale. Cependant, encore une fois, nous obtenons une **moyenne de 57 noeuds ouverts**, soit une moyenne supérieure à celle où nous utilisons la distance de Manhattan.

e) Ajout du lien de casse à la distance Euclidienne

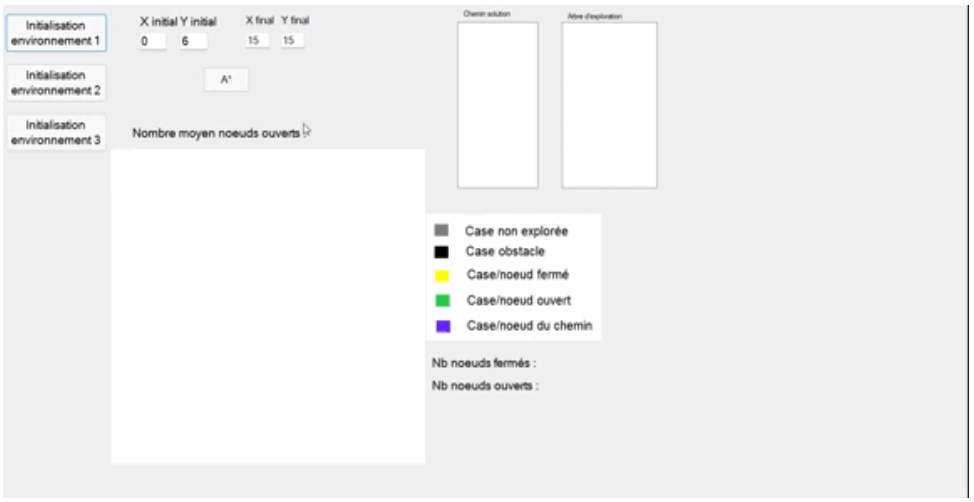


figure 5 : Vidéo de l'algorithme A* avec la distance euclidienne comme heuristique avec lien de casse

Suite à plusieurs tests, nous avons compris pourquoi l'heuristique distance de Manhattan était plus efficace que l'heuristique distance Euclidienne. Cela est dû au fait que dans le cas de l'heuristique distance Euclidienne, l'algorithme trouve plusieurs chemins au coût équivalent. De ce fait, il explore plus de chemins que nécessaire. Pour palier à ce problème, nous avons décidé d'ajouter un lien de casse. Un lien de casse permet d'augmenter légèrement la valeur de l'heuristique en le multipliant par un facteur p :

$$H(n) = H(n) + H(n) \times (1 + p)$$

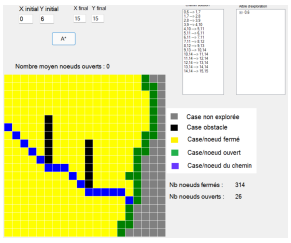
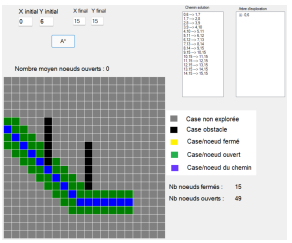
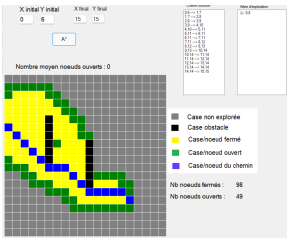
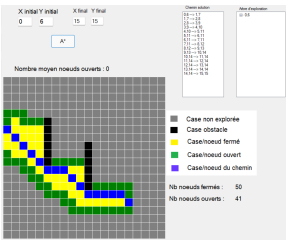
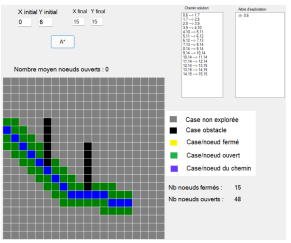
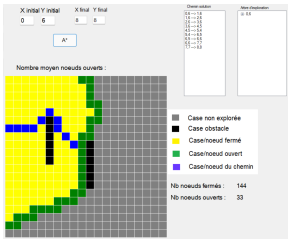
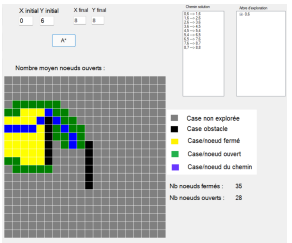
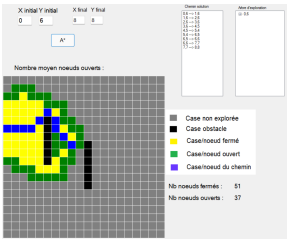
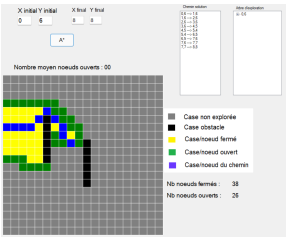
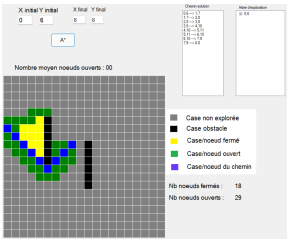
Pour choisir p, il faut choisir un nombre p tel que :

$$p < \frac{coût\ d'un\ pas\ minimum}{valeur\ maximale\ du\ chemin\ attendue}$$

Dans notre cas nous avons donc choisi p=0.01. Avec le lien de casse, nous obtenons ainsi une **moyenne de 39 nœuds ouverts**. Soit un nœud ouvert en moyenne de moins qu'avec l'heuristique distance de Manhattan.

A l'aide de nos tests, nous avons finalement réussi à réduire le nombre de nœuds ouverts en **moyenne de 179 nœuds.**

f) Tableau représentatif des différents cas selon l'heuristique

Dijsktra	Manhattan	Checbyshev	Euclidien sans lien de casse	Euclidien avec lien de casse
				
				

Au vu du tableau comparatif ci-dessus, nous remarquons que dans la situation initiale, les heuristiques de Manhattan et Euclidien sont quasiment équivalents, ils ne se différencient que par l'ouverture d'un noeud en plus pour l'heuristique de Manhattan. Cela correspond à ce que nous avons pu avoir dans notre moyenne sur 1000 tests. Pour le second test, l'heuristique Euclidien est cependant bien plus efficace que celui de Manhattan avec une différence de 41 nœuds ouverts.

Pour la suite du projet, ayant eu des résultats légèrement plus satisfaisants avec l'heuristique de distance euclidienne (une fois l'ajout du lien de casse établi) pour l'environnement 1, nous avons décidé de garder cette heuristique comme base de calcul de distance pour les environnements 1 et 2. Cependant, dans notre phase de recherche, nous effectuions tout de même les calculs avec le distance de Manhattan pour s'assurer de la validité de notre heuristique.

Environnement 2

Le code utilisé pour l'environnement 2 est disponible en annexe.

a) Algorithme de Dijkstra

figure 6 : Vidéo de l'algorithme de Dijkstra

$$H(n) = 0$$

Dans le cas de l'algorithme de Dijkstra, c'est-à-dire où l'heuristique est nul. Nous obtenons une **moyenne de 212 nœuds ouverts**.

b) Heuristique sans lien de casse

Pour l'environnement 2, nous avons différencié différentes situations et adapté l'heuristique en fonction.

De ce fait, si la position initiale et la position finale sont du même côté, c'est-à-dire à gauche ou à droite de l'obstacle, alors l'heuristique que nous appliquons est simplement un calcul de distance Euclidienne. Et dans le cas où la position finale se situe dans l'ouverture de l'obstacle, nous appliquons également un simple calcul de distance euclidienne :

$$H(n) = \sqrt{(xf - x)^2 + (yf - y)^2}$$

En revanche, si la position initiale et la position finale se trouvent des deux côtés (position initiale à gauche de l'obstacle et position à droite de l'obstacle par exemple), alors l'heuristique que nous appliquons est une somme de calculs de distance Euclidienne :

$$H(n) = \sqrt{(xi - x)^2 + (yi - y)^2} + \sqrt{(xf - xi)^2 + (yf - yi)^2}$$

xi : *Position du trou dans l'obstacle en abscisse*

yi : *Position du trou dans l'obstacle en ordonnée*

Ainsi, nous trouvons le chemin le plus court jusqu'au trou dans l'obstacle, puis du trou nous trouvons le chemin le plus court jusqu'à la position finale à atteindre.

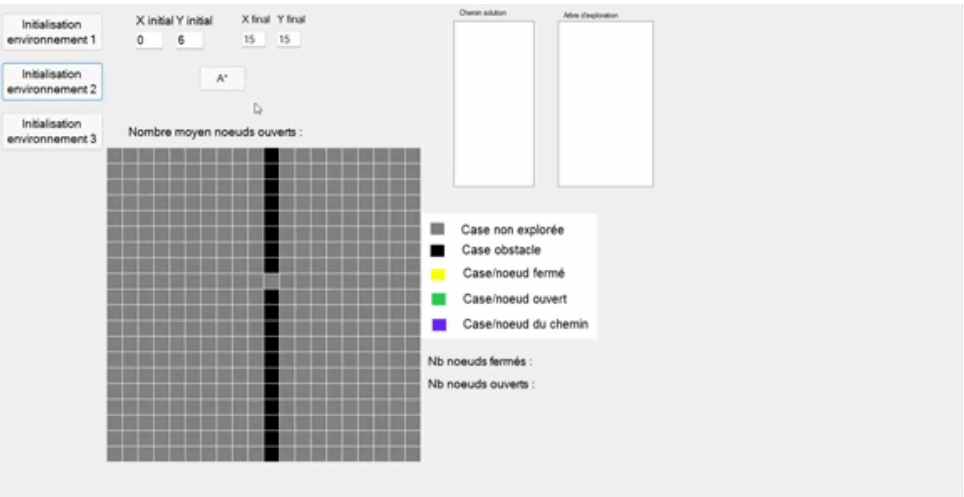


figure 7 : Vidéo de l'algorithme A* avec notre heuristique

A l'aide de cette heuristique, nous obtenons une **moyenne de 54 nœuds ouverts**, ce qui est déjà bien inférieur à la moyenne de nœuds ouverts avec l'algorithme de Dijkstra. Cependant, nous rencontrons le même problème qu'avec l'environnement 1. En effet, l'algorithme explore des chemins équivalents. Nous devons donc ajouter le lien de casse comme pour l'environnement 1.

c) Heuristique avec lien de casse

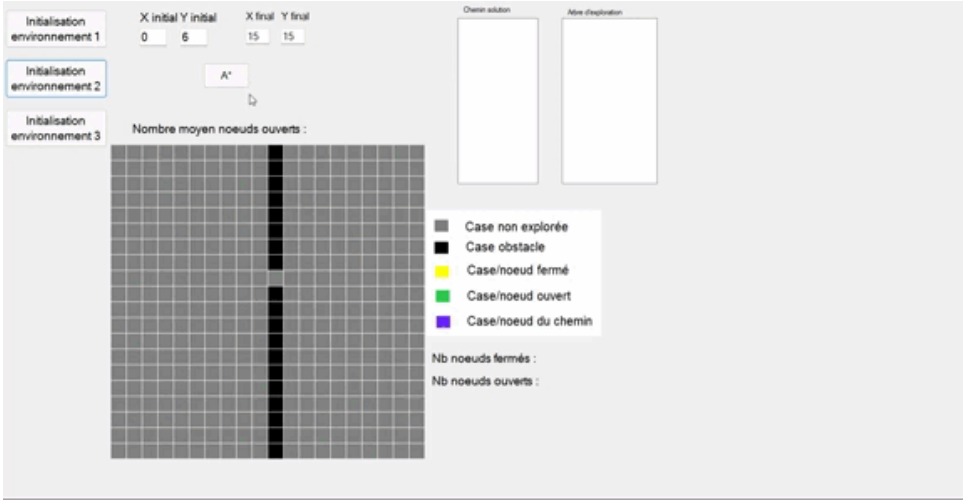
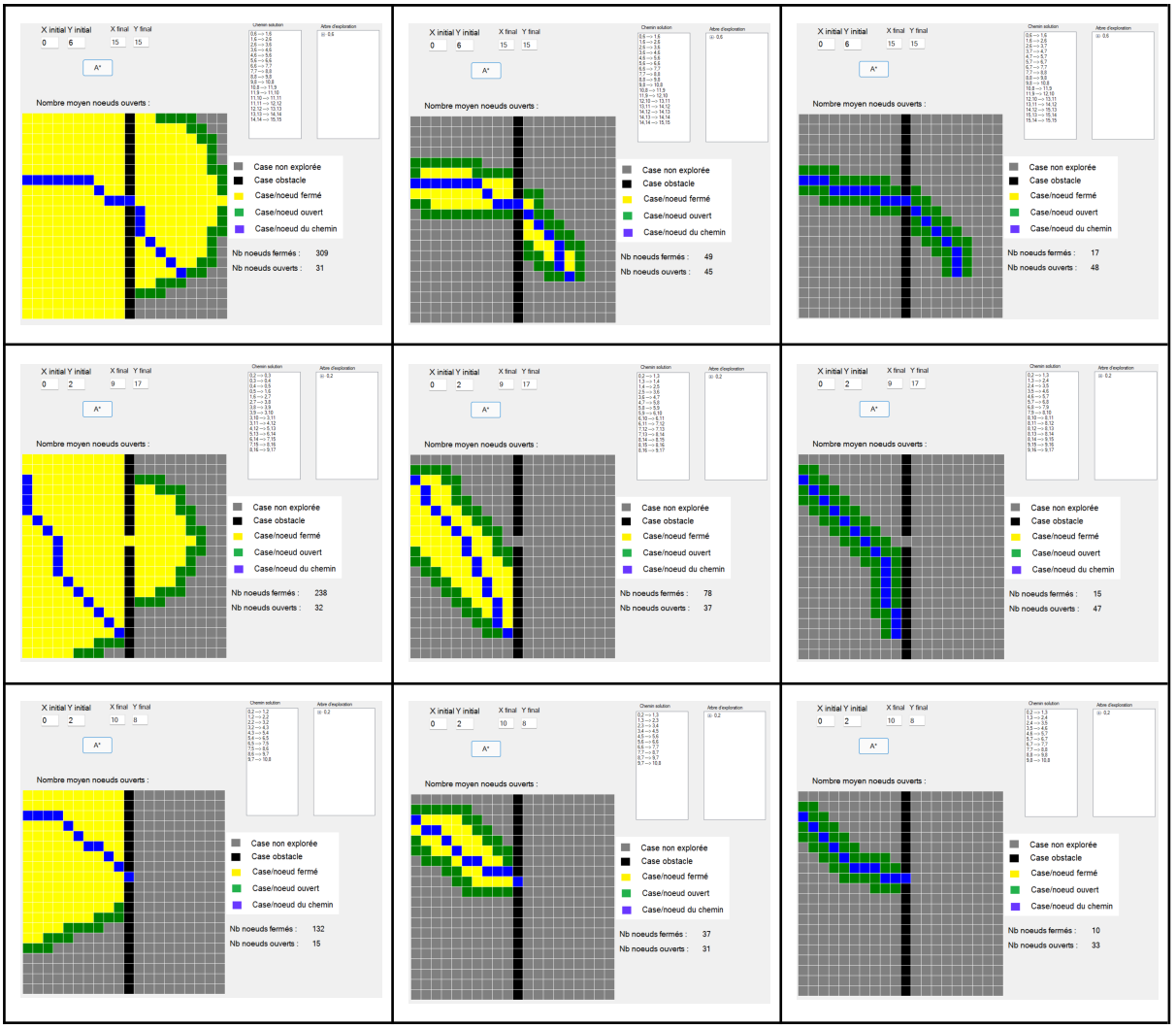


figure 7 : Vidéo de l'algorithme A* avec notre heuristique avec lien de casse

Suite à l'ajout du lien de casse, nous obtenons une **moyenne de 41 nœuds ouverts**. Nous avons également choisi p=0.01 ici pour le lien de casse. A l'aide de notre heuristique, nous réduisons donc en moyenne le nombre de nœuds ouverts de 171 nœuds par rapport à l'algorithme de Dijkstra.

d) Tableau représentatif des différents cas selon l'heuristique

Dijkstra	Euclidien sans lien de casse	Euclidien avec lien de casse



Dans le tableau ci-dessous sont illustrés les différents cas que nous avons énuméré précédemment. Comme nous pouvons le voir, notre heuristique permet de réduire le nombre de cases visitées. Une fois l'ajout du lien de casse, nous réduisons encore plus le nombre de cases visitées.

Environnement 3

a) Algorithme de Dijkstra

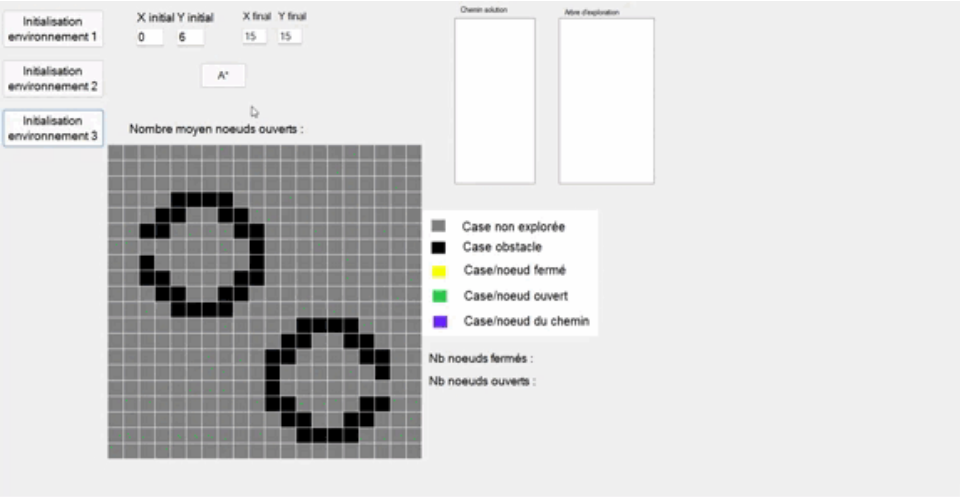


figure 8 : Vidéo de l'algorithme de Dijkstra

$$H(n) = 0$$

Dans le cas de l'algorithme de Dijkstra, c'est-à-dire où l'heuristique est nulle. Nous obtenons une **moyenne de 189 nœuds ouverts**.

b) Heuristique sans lien de casse

Comme pour l'environnement 2, nous avons ici différencié certains cas. En effet, les cas suivants sont envisageables :

- Le noeud actuel se trouve dans l'obstacle en haut à gauche
- Le noeud actuel se trouve dans l'obstacle en bas à droite
- Le noeud à atteindre se trouve dans l'obstacle en haut à gauche
- Le noeud à atteindre se trouve dans l'obstacle en bas à droite
- Le nœud actuel et le nœud à atteindre se trouvent en dehors des obstacles.

Pour savoir dans quel cas nous nous trouvions, nous avons créé deux fonctions *estDansObstacleHaut* et *estDansObstacleBas* qui nous indiquent si le noeud se trouve dans un des deux obstacles.

```
2 références
private bool estDansObstacleHaut(int x,int y)
{
    if ((y == 4) && ((x == 5) || (x == 6)))
        return true;
    else if ((y == 5) && ((x == 4)||(x == 5) || (x == 6)|| (x == 7)))
        return true;
    else if ((y == 6) && ((x == 2) || (x == 3)||(x == 4) || (x == 5) || (x == 6) || (x == 7) ||(x==8)))
        return true;
    else if ((y == 7) && ((x == 3) || (x == 4) || (x == 5) || (x == 6) || (x == 7) || (x == 8)))
        return true;
    else if ((y == 8) && ((x == 4) || (x == 5) || (x == 6) || (x == 7)))
        return true;
    else if ((y == 9) && ((x == 5) || (x == 6)))
        return true;
    return false;
}

2 références
private bool estDansObstacleBas(int x, int y)
{
    if ((y == 12) && ((x == 13) || (x == 14)))
        return true;
    else if ((y == 13) && ((x == 12) || (x == 13) || (x == 14) || (x == 15)))
        return true;
    else if ((y == 14) && ((x == 11) || (x == 12) || (x == 13) || (x == 14) || (x == 15) || (x == 16) ))
        return true;
    else if ((y == 15) && ((x == 11) || (x == 12) || (x == 13) || (x == 14) || (x == 15) || (x == 16) || (x == 17)))
        return true;
    else if ((y == 16) && ((x == 12) || (x == 13) || (x == 14) || (x == 15)))
        return true;
    if ((y == 17) && ((x == 13) || (x == 14)))
        return true;
    return false;
}
```

figure 9 : Code des fonctions estDansObstacleHaut() et estDansObstacleBas()

Maintenant que nous savons où se situent nos noeuds, nous pouvons implémenter le code en fonction des différents cas. Ainsi, nous obtenons le code suivant :

```
if (estXYDansObstacleHaut)
    heuristique += CalculEuclideanDistance2(2, 6);
else if (estXYDansObstacleBas)
    heuristique += CalculEuclideanDistance2(17, 15);
else if (estXfYfDansObstacleHaut)
{
    Form1.matrice[17, 15] = -1;
    heuristique += CalculEuclideanDistance2(2, 6);
}

else if (estXfYfDansObstacleBas)
{
    Form1.matrice[2, 6] = -1;
    heuristique += CalculEuclideanDistance2(17, 15);
}
heuristique = heuristique + CalculeEuclideanDistance(x,y,xFinal,yFinal);
heuristique *= (1 + (1 % 100));
return heuristique;
```

figure 10 : Code de la fonction CalculHCost() pour l'environnement 3

Les fonctions CalculeEuclideanDistance() et CalculEuclideanDistance() sont implémentées de la manière suivante :

```
6 références
private double CalculeEuclideanDistance(int xActuel, int yActuel, int xFinal, int yFinal)
{
    return Math.Sqrt(Math.Pow((xFinal - xActuel), 2) + Math.Pow((yFinal - yActuel), 2));
}

4 références
private double CalculEuclideanDistance2(int xIntermediaire, int yIntermediaire)
{
    return CalculeEuclideanDistance(this.x, this.y, xIntermediaire, yIntermediaire)
        + CalculeEuclideanDistance(xIntermediaire, yIntermediaire, Form1.xfinal, Form1.yfinal);
}
```

figure 11 : Code des fonctions CalculeEuclideanDistance() et CalculEuclideanDistance2()

Cas n°1 : *le noeud actuel se trouve dans l'obstacle en haut à gauche*

Dans ce cas-là, nous effectuons trois calculs de distance euclidienne. Le premier allant de la position du noeud actuel à la sortie de l'obstacle haut puis le second, du point (2,6) à la position du noeud final. Nous rajoutons ensuite cette valeur à la variable heuristique. Puis lorsque nous sortons des if/else, nous ajoutons à l'heuristique la valeur de la distance euclidienne de la position actuelle du nœud jusqu'à la position du nœud final.

Cas n°2 *: le noeud actuel se trouve dans l'obstacle en bas à droite*

Dans ce cas-là également, nous effectuons trois calculs de distance euclidienne. Le premier allant de la position du nœud actuel à la sortie de l'obstacle bas puis le second, du point (17,15) à la position du nœud final. Nous rajoutons ensuite cette valeur à la variable heuristique. Puis lorsque nous sortons des if/else, nous rajoutons à l'heuristique la valeur de la distance euclidienne de la position actuelle du noeud jusqu'à la position du noeud final.

Cas n°3 : *le noeud final se trouve dans l'obstacle en haut à gauche*

Si nous sommes dans ce cas, cela signifie que notre noeud actuel ne se trouve pas dans un des deux obstacles et que notre noeud final se trouve dans l'obstacle en haut à gauche. Nous pouvons donc nous permettre de fermer l'entrée de

l'obstacle en bas à droite pour éviter que l'algorithme n'explore ces noeuds. Puis après, nous effectuons trois calculs de distance euclidienne. Le premier allant de la position du noeud actuel à l'entrée de l'obstacle haut puis le second, de l'entrée de l'obstacle à la position du noeud final. Nous rajoutons ensuite cette valeur à la variable heuristique. Puis lorsque nous sortons des if/else, nous rajoutons à l'heuristique la valeur de la distance euclidienne de la position actuelle du noeud jusqu'à la position du noeud final.

Cas n°4 : *le noeud final se trouve dans l'obstacle en bas à droite*

Si nous sommes dans ce cas, cela signifie que notre noeud actuel ne se trouve pas dans un des deux obstacles et que notre noeud final se trouve dans l'obstacle en bas à gauche. Nous pouvons donc nous permettre de fermer l'entrée de l'obstacle en haut à gauche pour éviter que l'algorithme n'explore ces noeuds. Puis, nous effectuons trois calculs de distance euclidienne. Le premier allant de la position du noeud actuel à l'entrée de l'obstacle bas puis le second, de l'entrée de l'obstacle à la position du noeud final. Nous rajoutons ensuite cette valeur à la variable heuristique. Puis lorsque nous sortons des if/else, nous rajoutons à l'heuristique la valeur de la distance euclidienne de la position actuelle du noeud jusqu'à la position du noeud final.

Cas n°5 : *le noeud actuel et le noeud à atteindre se trouvent en dehors des obstacles*

Dans ce cas-là, nous ne rentrons pas dans les boucles if/else. Nous effectuons ainsi un simple calcul de distance euclidienne de la position du noeud actuel jusqu'à la position du noeud final.

Après avoir implémenté notre heuristique, nous avons, comme pour les environnements 1 et 2 effectué 1000 tests et calculé la moyenne.

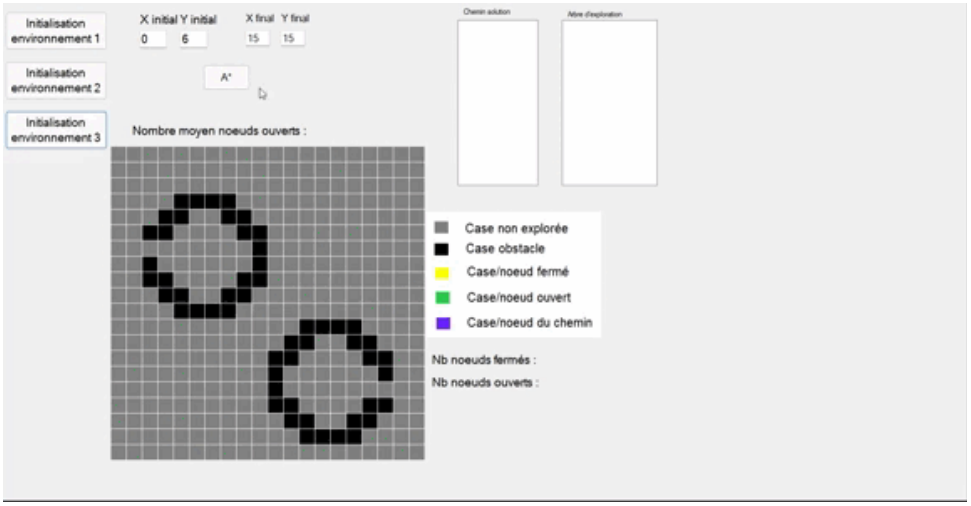


figure 11 : Vidéo de l'algorithme A* avec notre heuristique sans lien de casse

A l'aide de cette heuristique, nous obtenons une **moyenne de 66 nœuds ouverts**, ce qui est déjà bien inférieur à la moyenne de nœuds ouverts avec l'algorithme de Dijkstra. Cependant, nous rencontrons le même problème qu'avec les environnements 1 et 2. En effet, l'algorithme explore des chemins équivalents. Nous devons donc ajouter le lien de casse comme pour les environnements 1 et 2.

c) Heuristique avec lien de casse

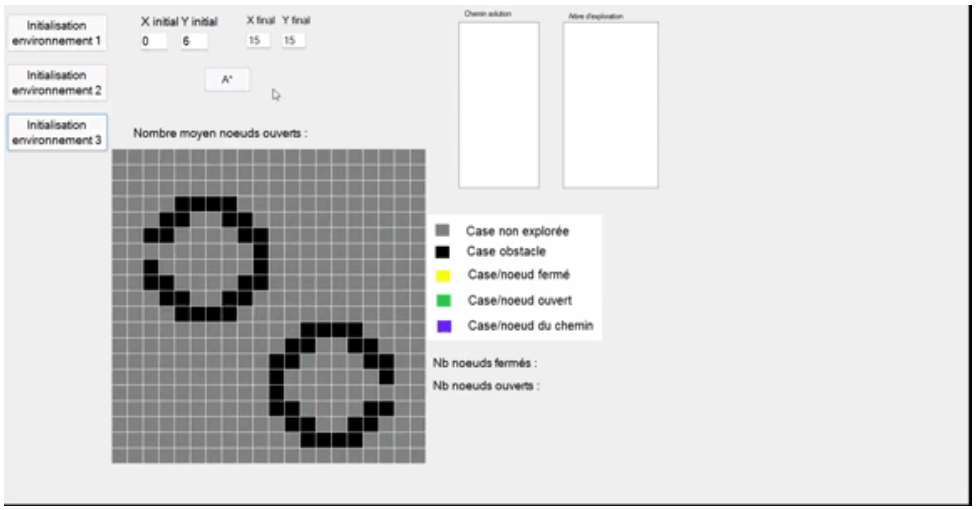
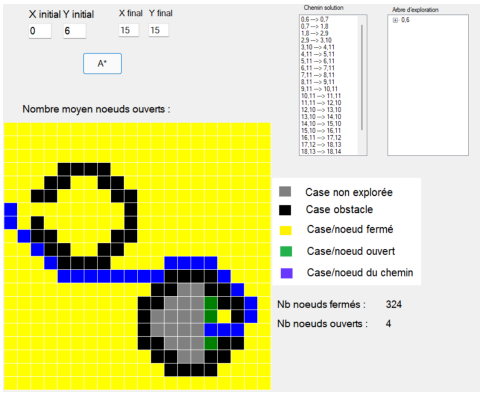
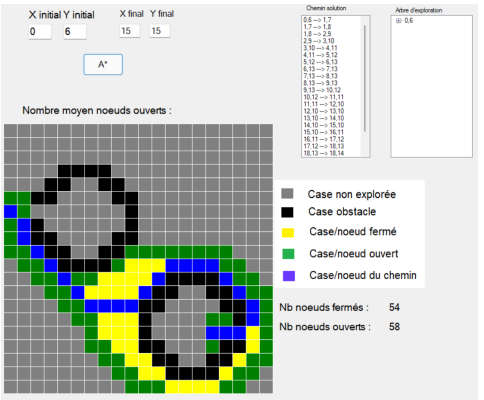
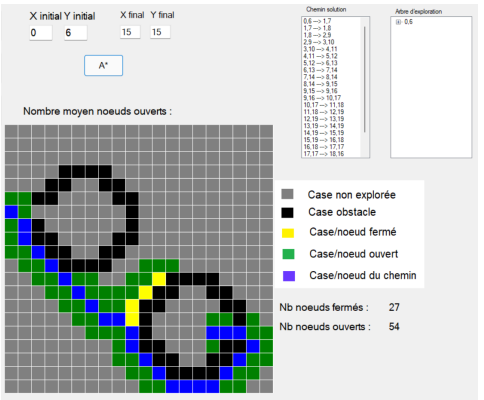
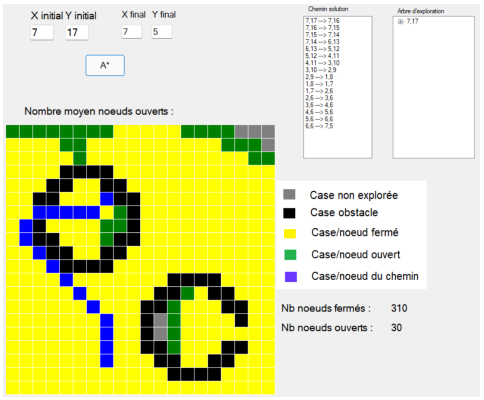
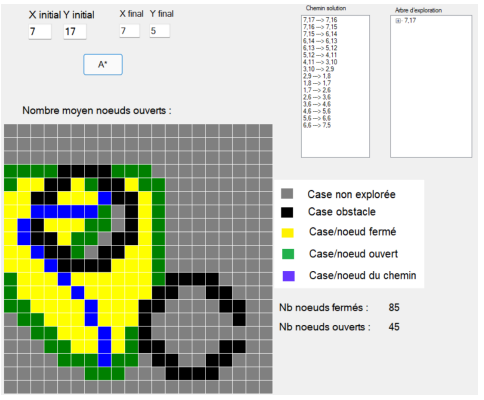
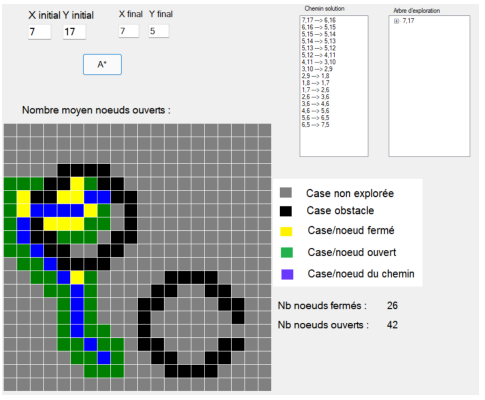
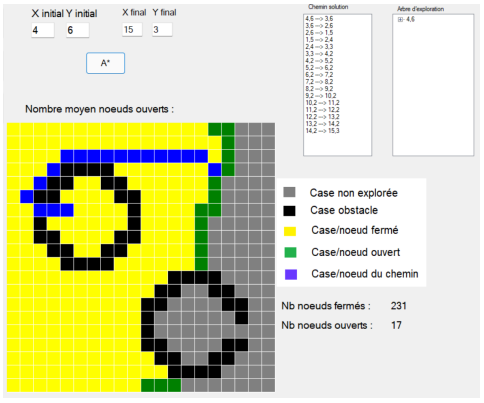
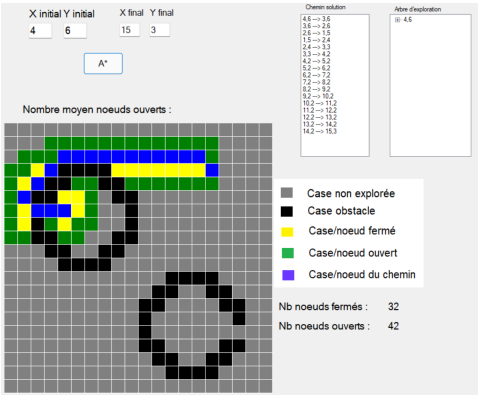
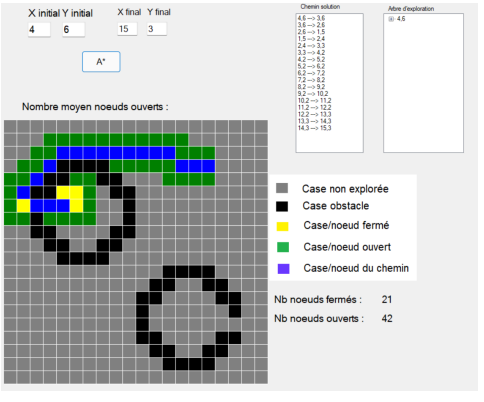
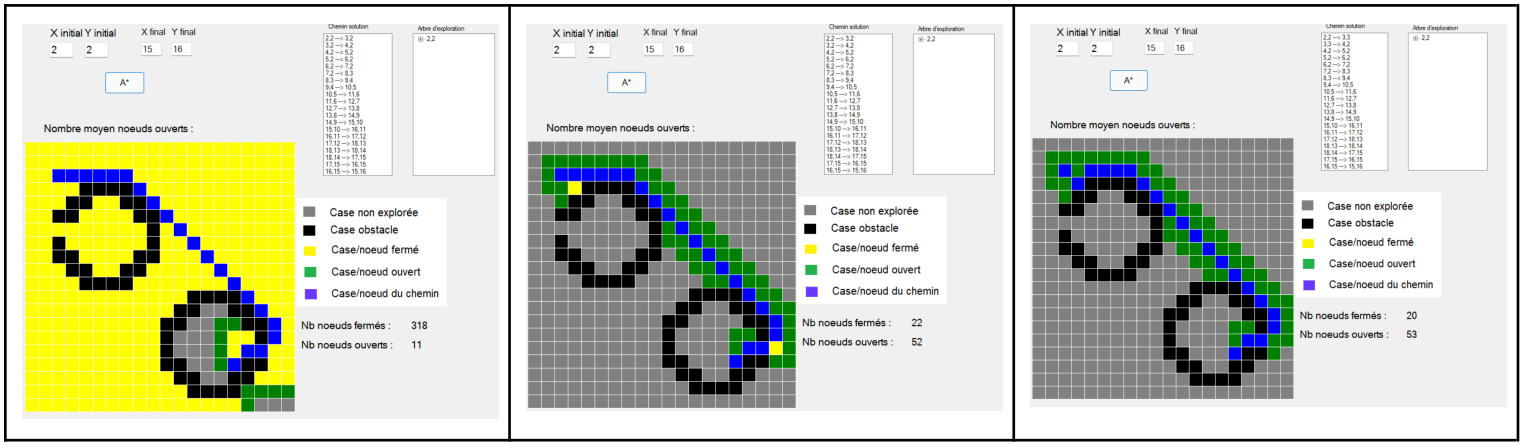


figure 12 : Vidéo de l'algorithme A* avec notre heuristique avec lien de casse

Suite à l'ajout du lien de casse, nous obtenons une **moyenne de 46 nœuds ouverts** sur 1000 tentatives. Nous avons également choisi p=0.01 ici pour le lien de casse. A l'aide de notre heuristique, nous réduisons donc en moyenne le nombre de nœuds ouverts de 143 nœuds par rapport à l'algorithme de Dijkstra.

e) Tableau représentatif des différents cas selon l'heuristique

Dijkstra	Heuristique sans lien de casse	Heuristique avec lien de casse
		
		
		



Dans le tableau ci-dessus sont représentés les différents cas que nous avions énuméré précédemment. Comme nous pouvons le voir, notre heuristique permet de réduire drastiquement le nombre noeuds visités. Nous observons également que lorsqu'un noeud final se trouve dans un obstacle et que le noeud actuel est en dehors d'un obstacle, l'algorithme n'explore pas l'obstacle vide (où le noeud final ne se situe pas).

Conclusion

Ce premier projet d’IA nous a permis de nous familiariser avec de nouvelles philosophies à adopter face à la résolution de problèmes. Nous avons réussi à aboutir à des résultats concluants, que ce soit pour les temps d'exécution, qui ne s’éternisent pas trop, du programme, mais aussi pour les temps de parcours de la course et les nombres de nœuds ouverts et fermés à la fin des calculs.

Annexe

```
if (Form1.environnementLoaded == "environnement1.txt")
{
    //Algorithme de Dijsktra
    //return 0;

    //Heuristique Manhattan distance
    //heuristique = Manhattan(xFinal, yFinal);
    //return heuristique;


    //Heuristique distance diagonale
    //heuristique = 1 * (Math.Abs(this.x - xFinal) + Math.Abs(this.y - yFinal)) + (1 - 2 * 1) * Math.Min(Math.Abs(this.x - xFinal), Math.Abs(this.y - yFinal));
    //return heuristique;

    //Heuristique Euclidean distance
    heuristique = CalculeEuclideanDistance(x, y, xFinal, yFinal);
    heuristique *= (1 + (1 % 100));
    return heuristique;

}
```

Annexe 1 : Code de la fonction CalculeHCost() pour l'environnement 1

```
else if (Form1.environnementLoaded == "environnement2.txt")
{
   //Algorithme de Dijsktra
   //return 0;

   if ((x < 10 && xFinal < 10) || (x > 10 && xFinal > 10))
   {
       heuristique = CalculeEuclideanDistance(x, y, xFinal, yFinal);
       //heuristique = dx + dy;
       heuristique *= (1 + (1 % 100));
       return heuristique;
   }

   else if ((x < 10 && xFinal > 10) || (x > 10 && xFinal < 10))
   {
       heuristique = CalculeEuclideanDistance(x, y, xMilieuEnvironnement2, yMilieuEnvironnement2) + CalculeEuclideanDistance(xMilieuEnvironnement2, yMilieuEnvironnement2, xFinal, yFinal);
       //heuristique = dx + dy;
      heuristique *= (1 + (1 % 100));
       return heuristique;
   }
   heuristique = CalculeEuclideanDistance(x,y,xFinal,yFinal);
   heuristique *= (1 + (1 % 100));
   return heuristique;
}
```

Annexe 2 : Code de la fonction CalculeHCost() pour l'environnement 2

```
else
{
    bool estXYDansObstacleHaut = estDansObstacleHaut(x, y);
    bool estXfYfDansObstacleHaut = estDansObstacleHaut(xFinal, yFinal);
    bool estXYDansObstacleBas = estDansObstacleBas(x, y);
    bool estXfYfDansObstacleBas = estDansObstacleBas(xFinal, yFinal);

    //CODE Manhattan
    /*if (estXYDansObstacleHaut)
         heuristique +=  10*Manhattan2(2, 6);
     else if (estXYDansObstacleBas)
         heuristique +=  10*Manhattan2(17, 15);
     else if (estXfYfDansObstacleHaut)
    {
         Form1.matrice[17, 15] = -1;
         heuristique +=  10*Manhattan2(2, 6);
    }
     else if (estXfYfDansObstacleBas)
     {
        Form1.matrice[2, 6] = -1;
         heuristique +=  10*Manhattan2(17, 15);
     }

     heuristique = heuristique + Manhattan(xFinal,yFinal);
     heuristique*=(1+(1%100));
     return heuristique + Manhattan(xFinal, yFinal);*/

    //CODE EUCLIDEAN
    if (estXYDansObstacleHaut)
        heuristique += CalculEuclideanDistance2(2, 6);
    else if (estXYDansObstacleBas)
        heuristique += CalculEuclideanDistance2(17, 15);
    else if (estXfYfDansObstacleHaut)
    {
        Form1.matrice[17, 15] = -1;
        heuristique += CalculEuclideanDistance2(2, 6);
    }

    else if (estXfYfDansObstacleBas)
    {
        Form1.matrice[2, 6] = -1;
        heuristique += CalculEuclideanDistance2(17, 15);
    }
    heuristique = heuristique + CalculeEuclideanDistance(x,y,xFinal,yFinal);
    heuristique *= (1 + (1 % 100));
    return heuristique;
}
```

Annexe 3 : Code de la fonction CalculeHCost() pour l'environnement 3

```
1 référence
private void button2_Click(object sender, EventArgs e)
{
    int nbMoyenNoeuds = 0;
    for (int k = 0; k < 1000; k++)
    {
        load_environment(environnementLoaded);
        xinitial = Convert.ToInt32(textBoxXInit.Text);
        yinitial = Convert.ToInt32(textBoxYinit.Text);
        xfinal = Convert.ToInt32(textBoxXFinal.Text);
        yfinal = Convert.ToInt32(textBoxYFinal.Text);

       xinitial = xInitial[k];
        yinitial = yInitial[k];
        xfinal = xFinal[k];
        yfinal = yFinal[k];

        textBoxXInit.Text = Convert.ToString(xinitial);
        textBoxYinit.Text = Convert.ToString(yinitial);
        textBoxXFinal.Text = Convert.ToString(xfinal);
        textBoxYFinal.Text = Convert.ToString(yfinal);

        SearchTree search = new SearchTree();
        Class1 N0 = new Class1();
        N0.x = xinitial;
        N0.y = yinitial;
        List<GenericNode> solution = search.RechercheSolutionAEtoile(N0);

        // Affichage de l'arbre d'exploration
        search.GetSearchTree(treeView1);

        // Affichage des noeuds explorés
        // Affichage dans le picture box
        SolidBrush brush1 = new SolidBrush(Color.Yellow);
        SolidBrush brush2 = new SolidBrush(Color.Green);
        SolidBrush brush3 = new SolidBrush(Color.Blue);
        int largeur = pictureBox1.Width / nbcolonnes;
        int hauteur = pictureBox1.Height / nbcolonnes;
        Rectangle rect;
        // Les fermés
        for (int i = 0; i < search.L_Fermes.Count; i++)
        {
            Class1 noeudferme = (Class1)search.L_Fermes[i];
            rect = new Rectangle(noeudferme.x * largeur, noeudferme.y * hauteur, largeur - 1, hauteur - 1);
            g.FillRectangle(brush1, rect);
        }
        // Les ouverts
        for (int i = 0; i < search.L_Ouverts.Count; i++)
        {
            Class1 noeudouvert = (Class1)search.L_Ouverts[i];
            rect = new Rectangle(noeudouvert.x * largeur, noeudouvert.y * hauteur, largeur - 1, hauteur - 1);
            g.FillRectangle(brush2, rect);
        }

        // Affichage de la solution en bleu
        listBox1.Items.Clear();
        Class1 N1 = N0;
        rect = new Rectangle(N1.x * largeur, N1.y * hauteur, largeur - 1, hauteur - 1);
        g.FillRectangle(brush3, rect);
        for (int i = 1; i < solution.Count; i++)
        {
            Class1 N2 = (Class1)solution[i];
            listBox1.Items.Add(Convert.ToString(N1.x) + "," + Convert.ToString(N1.y)
                + " ---> " + Convert.ToString(N2.x) + "," + Convert.ToString(N2.y));
            rect = new Rectangle(N2.x * largeur, N2.y * hauteur, largeur - 1, hauteur - 1);
            g.FillRectangle(brush3, rect);
            N1 = N2;
        }
        // Affichage du nb de noeuds dans ouverts et fermés
        labelOuverts.Text = Convert.ToString(search.L_Ouverts.Count);
        labelFermes.Text = Convert.ToString(search.L_Fermes.Count);
        nbMoyenNoeuds += (search.L_Ouverts.Count + search.L_Fermes.Count);

        //Pour vérifier que y'ait pas de coordonnées générée sur un obstacle
        if (matrice[xinitial, yinitial] == -1 || matrice[xfinal, yfinal] == -1)
            Console.WriteLine("({0},{1}) : ({2},{3})", xinitial, yinitial, xfinal, yfinal);
    }
    label9.Text += Convert.ToString(nbMoyenNoeuds / 1000);
    Console.WriteLine(nbMoyenNoeuds / 1000);
}
```

Annexe 4 : Code de la fonction button2_Click() pour la boucle