

BARBAT Hadrien  
CATELLA Solène  
ROSALIE Rodolphe

14/04/2017

# **PROJET I.A.**

ENSC - 2017

# **Partie I**

## **Question 1 : choix du plus court chemin**

### **Choix effectués :**

Pour l'algorithme du plus court chemin, nous avons choisi de représenter l'entrepôt par une grille [25,25] d'entiers.

Ainsi, les différents états (ou valeurs) possibles pour chacune des cases sont les suivants :

- 0 pour une case vide,
- 1 pour désigner une case étagère,
- 2 pour désigner une case zone de déchargement,
- 3 pour désigner une case marchandise,
- 4 pour désigner une case où un chariot se trouve.

Ces valeurs permettent de connaître non seulement la disponibilité d'une case, mais aussi de dessiner les cases spécifiques de l'entrepôt avec une couleur particulière.

- Une classe Position a été créée : le premier constructeur permettant de stocker les coordonnées (x, y) de chaque position d'un chariot, le second de lui ajouter une orientation.
- Deux méthodes, Equals et Equals 2, permettent de comparer des positions.
- La classe NodeChariotChemin forme la structure de nos nodes. Un node est uniquement constitué de la position actuelle (x,y) du chariot. Deux autres paramètres sont initialisés via la fonction statique initialiserTout : la position actuelle, pour pouvoir la mettre à jour, la position finale (c'est-à-dire la destination) du chariot, ainsi que la grille représentant l'entrepôt.
- La fonction GetListSucc permet au chariot d'explorer la grille dans les différentes directions, en évitant les directions interdites : un chariot ne peut circuler que sur une case libre, dont la valeur est 0.
- Le formulaire associé Form2 permet de dessiner l'entrepôt ainsi que le meilleur chemin calculé.

Le chemin final est trouvé lorsque la position de la case actuellement explorée correspond à la position de destination.

### **Tests et heuristiques :**

Deux tests différents ont été effectués : l'un avec heuristique, l'autre sans.

L'heuristique mise en place vise à calculer le chemin le plus court "à vol d'oiseau", c'est-à-dire qu'à chaque fois que le chariot avance, on regarde dans l'ensemble de ses déplacements possibles la case la plus proche de la destination en termes de distance euclidienne.

Voici les résultats obtenus (le point de départ est en bas à gauche) :

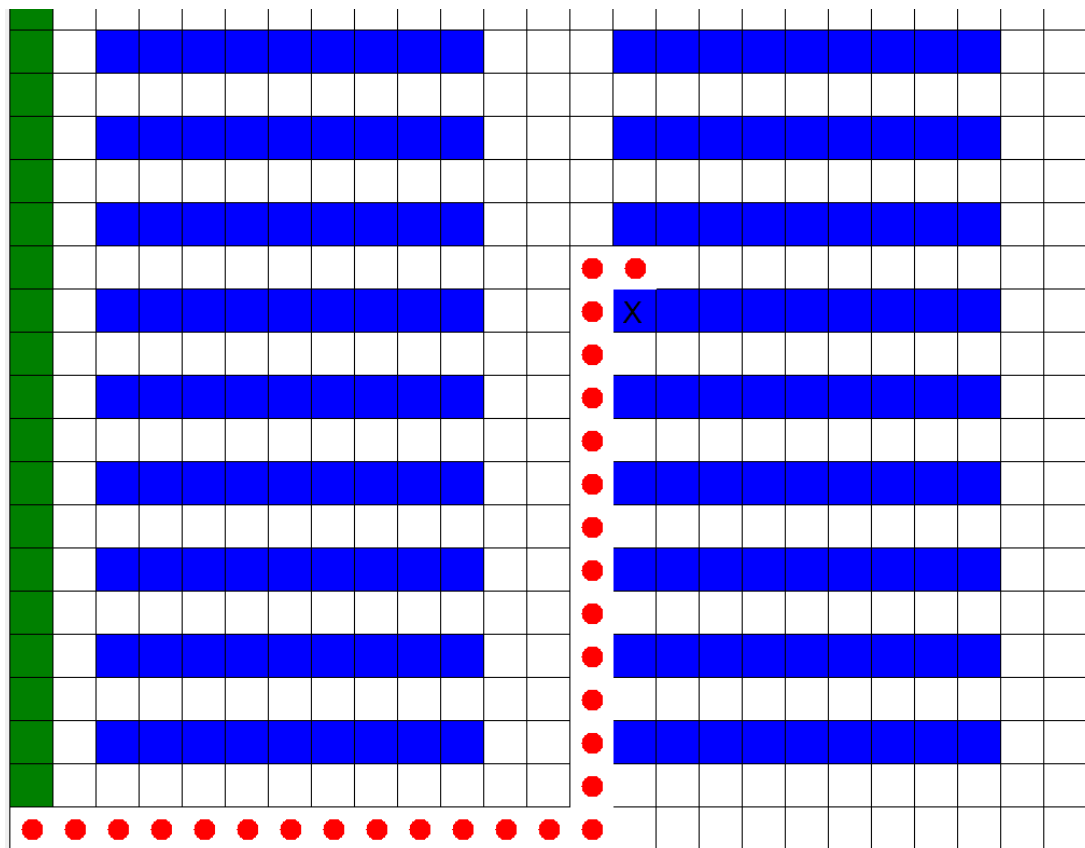


Figure sans heuristique : 13 nodes ouverts - 254 fermés

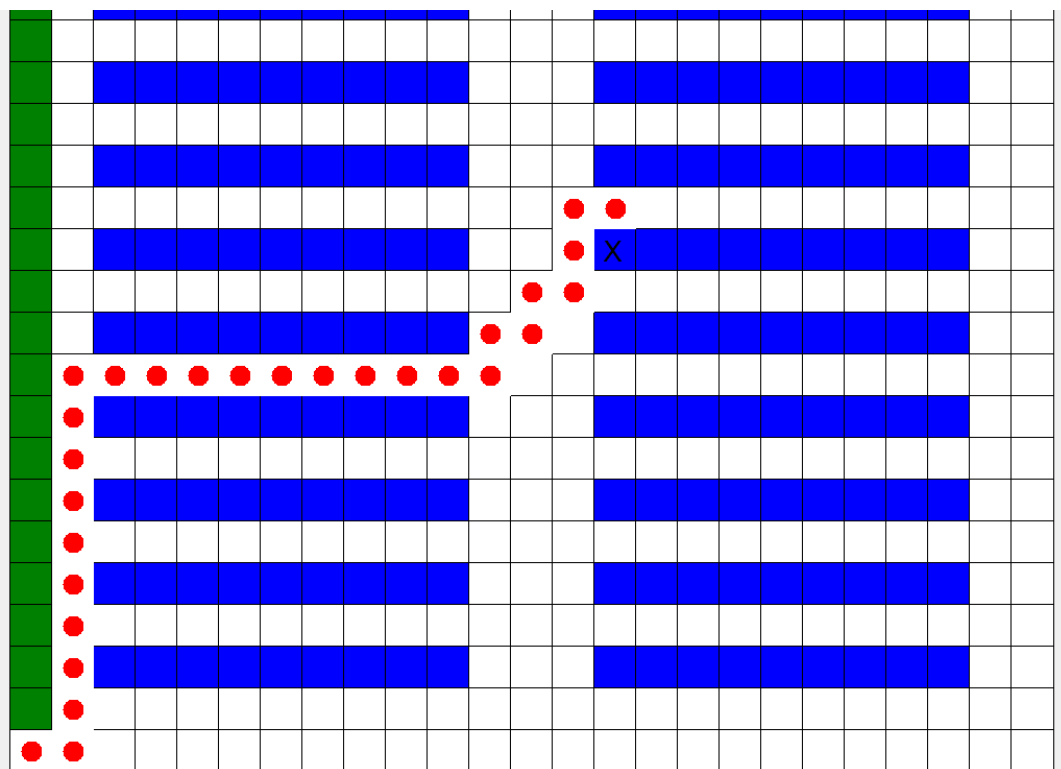


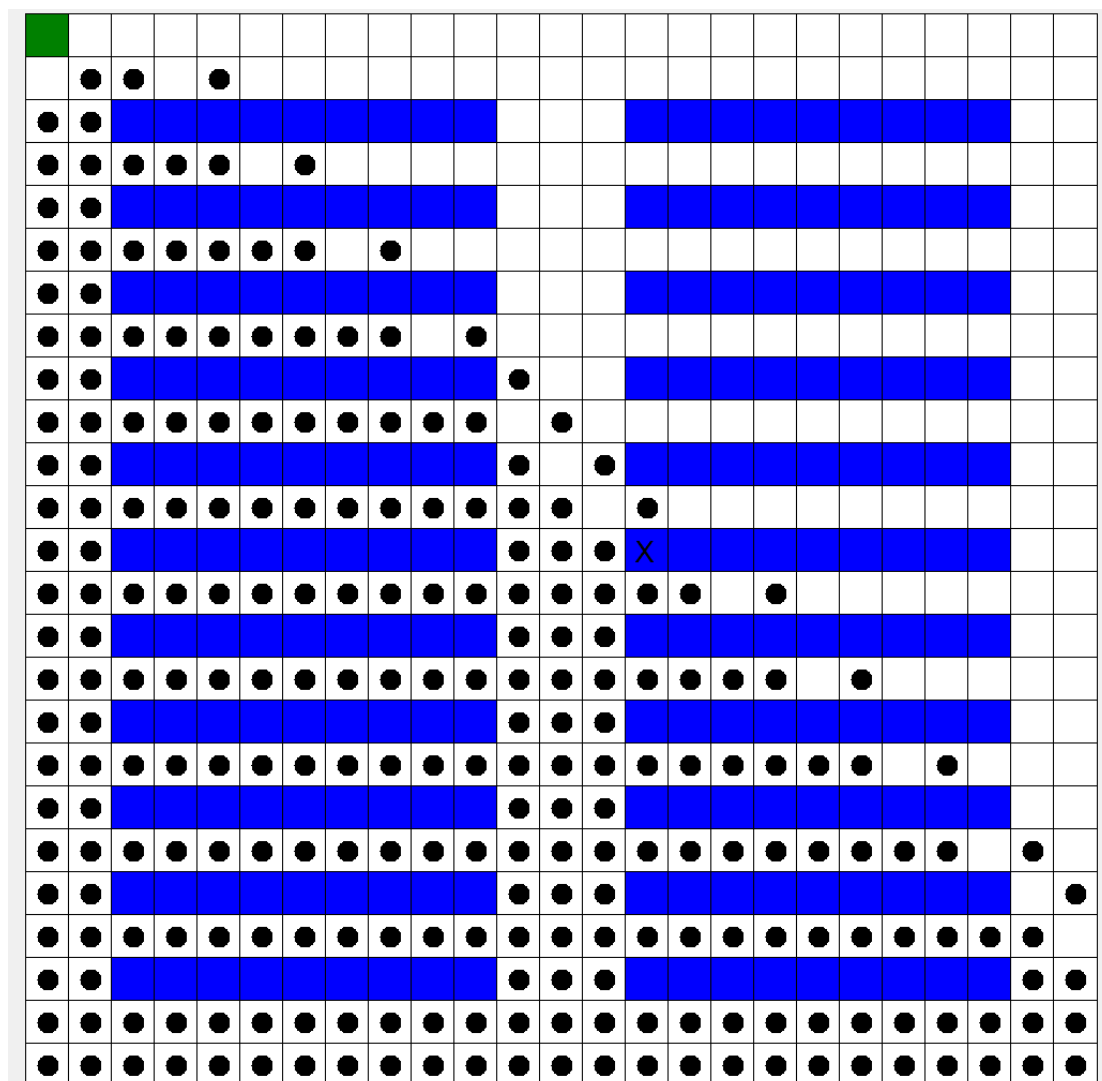
Figure avec heuristique : 15 nodes ouverts - 141 fermés

### Commentaires :

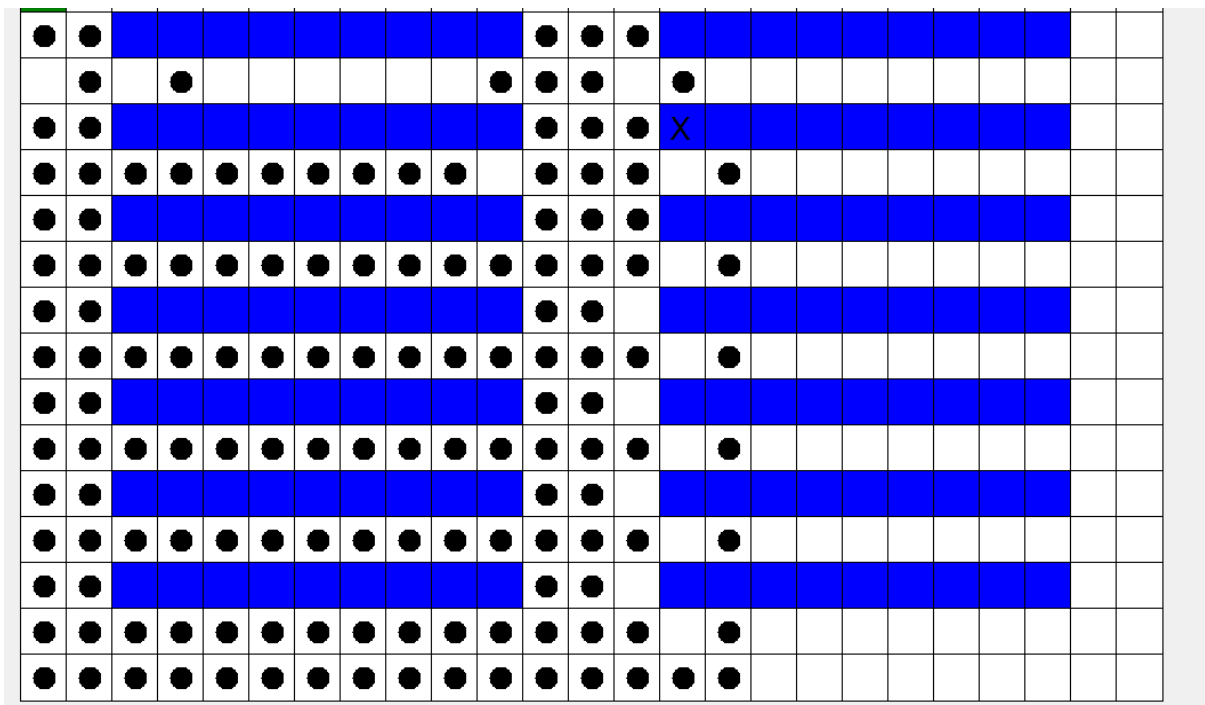
Un véritable changement de trajectoire est observé.

Avec l'heuristique, le chemin établi est directement orienté vers le point de marchandise. La diagonale formée par les points au centre de l'entrepôt témoigne bien de la recherche du plus court chemin "à vol d'oiseau". De plus, le nombre de nodes fermés diminue, ce qui témoigne bien d'une recherche plus efficace avec l'heuristique.

Les points noirs des figures ci-dessous montrent les cases explorées sans et avec heuristique. Le gain de performance avec heuristique est notable dans la mesure où la zone explorée est dans ce cas de figure bien moins étendue.



Zone explorée sans heuristique



Zone explorée avec heuristique

## **Question 2 : détermination d'une trajectoire spatio-temporelle**

### **Choix effectués et heuristique utilisée :**

L'algorithme utilisé pour déterminer la meilleure trajectoire spatio-temporelle reprend la grille [25,25] et les valeurs définissant l'état de chaque case.

- La classe NodeChariotTemps définit nos nodes.

Un node est composé d'une position (les coordonnées x et y ainsi que l'orientation actuelle du chariot : 1, 2, 3 ou 4) et d'un coût :

- 1 si l'orientation du chariot est la même lors de l'avancée d'une case à l'autre,
- 3 si une rotation (bifurcation) est nécessaire.

Plusieurs autres paramètres permettent de gérer le comportement de la recherche, et sont là encore initialisés avec InitialiserTout().

- La position finale correspond dans un premier temps à la marchandise à aller chercher, et dans un second temps à la zone de déchargement la plus proche.
- La hauteur de la marchandise permet de connaître le temps que mettra le chariot à récupérer l'objet.
- La grille, représentant l'entrepôt, est dans cette classe statique.
- Un booléen, charge, permet de savoir si le chariot est chargé ou non.

- Enfin, une liste de positions `mesCollisions` stocke les positions des collisions éventuellement détectées (avec d'autres chariots) lors de l'exploration de l'entrepôt. La fonction `instancieCollisions()` permet d'initialiser cette liste du formulaire.

Pour le cas des trajectoires spatio-temporelles, le calcul des trajectoires est effectué en deux temps. Le chariot doit d'abord aller chercher la marchandise : c'est sa première destination. Ensuite, le chemin de retour à la zone de déchargement est dans un second temps calculé. Dans chaque cas, la fonction `EndState()` nous permet de savoir si l'algorithme a trouvé un chemin jusqu'à destination. Si la position actuelle du chariot correspond à la position de la marchandise, la première étape est validée : le booléen `charge` est mis à `true`. Le chemin-retour vers la zone de déchargement la plus proche peut alors être calculé. Ce calcul est effectué via la fonction `PlusCourtRetour()`. Dans cette fonction, la zone de déchargement libre la plus proche est déduite d'après le calcul de la distance euclidienne. On vérifie donc qu'il n'y ait pas de chariot à cet emplacement, mais également qu'il n'y ait pas de collision possible sur le trajet.

`GetListSucc()` est de nouveau utilisé pour l'exploration de l'entrepôt. En plus d'ajouter les positions à la liste `listeGenericNode`, on veille également à transmettre le coût du mouvement vers cette position. Ce coût est calculé à chaque déplacement. Si l'orientation actuelle ne correspond pas au sens du déplacement, on change cette orientation : le coût est alors de 3. Dans le cas contraire, le coût est de 1.

- La classe comporte également la fonction `contient()`, déclinée sous deux versions :
  - la première prend en paramètre une liste de `NodeChariotTemps` et un index. Elle permet de vérifier que le node actuel n'est pas contenu dans la liste, et retourne sa position dans le cas contraire. Elle offre donc un moyen de détecter les collisions.
  - la seconde prend en paramètre uniquement un `NodeChariotTemps` et vérifie leurs égalités. Elle permet de gérer les collisions aux positions finales (sur la zone de déchargement) avec d'autres chariots déjà présents.

L'appel de ces deux fonctions est effectué dans le formulaire. Ce dernier s'est largement complexifié du fait de la prise en compte de la dimension spatio-temporelle et de la gestion simultanée de plusieurs chariots. Pour les tests de développements, 5 chariots ont été créés. Une liste `mesChariots` permet de contenir ces chariots. La liste `listeCollisions` stocke les collisions éventuellement détectées lors du déroulement de l'algorithme. Le tableau `mesChemins` contient des listes, permettant de stocker les différents chemins, de les parcourir et de les comparer.

Bien que des chariots ont été codés en durs, l'entrée et le positionnement peuvent se faire manuellement grâce au formulaire `InitialiseForm`. Dans ce cas, c'est la fonction `initialiseChariot()` qui récupère les coordonnées entrées et crée les chariots en conséquence. Les fonctions `InitialiseTab()` et `dessinTab()` permettent d'initialiser la grille et le dessin de l'entrepôt.

Une fois les chariots placés, à l'appui du bouton Go, l'algorithme de recherche est lancé. Les chariots sont traités de façon sérielle : les chemins sont recherchés chariot après chariot, et s'adaptent aux trajectoire précédemment calculées. Ainsi, le chariot 1 n'entrera jamais en collision avec d'autres chariots. En revanche, dès la deuxième itération de l'algorithme (recherche du chemin pour le deuxième chariot), est vérifié qu'il n'y ait pas de collision possible avec un autre chariot. La fonction Compare() va en effet comparer le chemin en cours de calcul à tous les chemins précédents. C'est dans celle-ci qu'on fait appel aux fonctions contient(). Compare() vérifie donc à la fois les éventuelles collisions sur les chemins, mais également au niveau des zones de déchargement.

Un problème algorithmique non élucidé nous a cependant contraints à créer la fonction TrouverZonesInterdites() pour pouvoir éviter les collisions dans les zones de chargement. De plus, la partie de la fonction Compare() censée prendre en charge cette fonctionnalité ne peut pas être effacée, sous peine de ne plus détecter les collisions.

Si une collision est détectée, on l'ajoute à la liste de collisions. La grille est réinitialisée, avec la valeur 4 aux emplacements où ont été détectées les collisions. L'algorithme de recherche recommence, avec la prise en compte des collisions : le chariot ne pourra pas se déplacer sur ces cases.

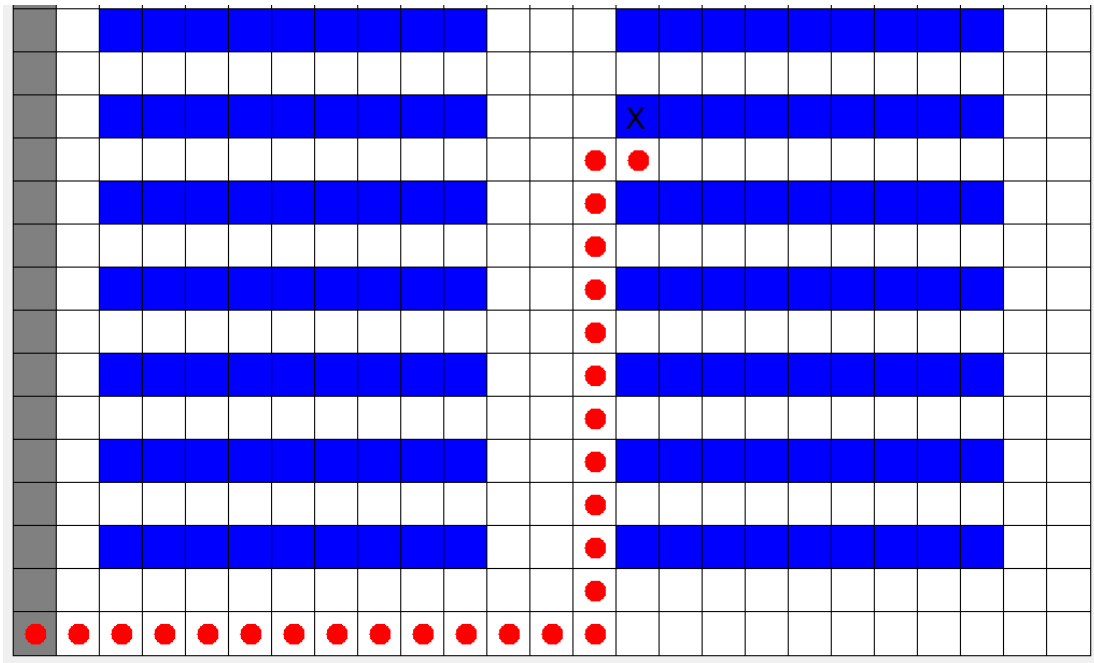
Si aucune collision n'est détectée à ce stade de l'algorithme, un chemin est établi : on passe au chariot suivant, puis on lance le dessin de l'avancée du dernier chariot via dessinAvanceChariot(). Avant d'avoir récupéré la marchandise, les chariots sont rouges ; une fois le chargement effectué, ils deviennent verts.

La fonction lance un thread pour chaque chariot qui sera dessiné, ce qui permet de les rendre indépendants. À chaque avancée, la case précédente est effacée, donnant ainsi une illusion de mouvement.

### **Tests :**

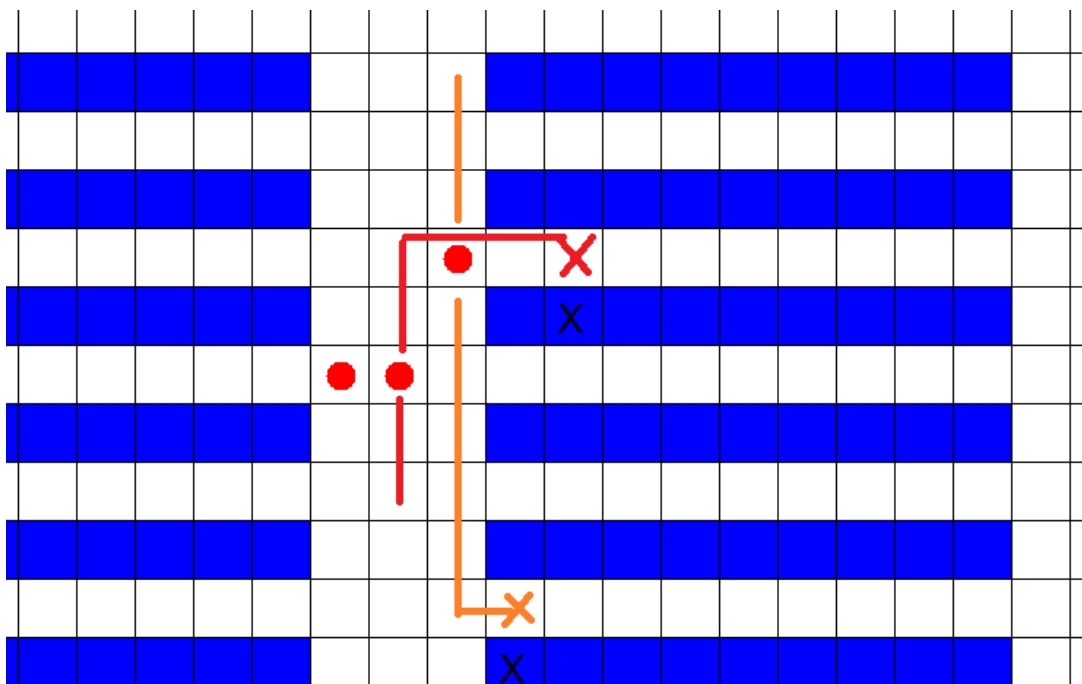
Plusieurs tests ont été effectués :

- Avec un chariot d'abord, pour comparer ce nouvel algorithme avec les chemins trouvés avec les NodesChariotsTemps.



Comme on peut le voir, pour une même position de départ et d'arrivée, la trajectoire définie est bien différente. Cela est due à la contrainte de coût, plus importante lors d'un tournant. Elle rend la trajectoire plus rectiligne, et donc avec moins de bifurcations.

- Nous avons ensuite testé l'algorithme avec plusieurs chariots. Les évitements sont bien fonctionnels.  
Voici un comportement d'évitement repéré entre deux chariots, avec leurs trajectoires en couleur rouge et orange :





Avec un algorithme ne prenant pas en compte les collisions, le chemin rouge rencontrerait le chemin orange. Lorsque l'on tient compte des collisions, le chemin rouge inclut un détour de façon à contourner la trajectoire orange.

Plusieurs bugs graphiques viennent cependant perturber le dessin de l'avancée des chariots. Tout d'abord, nous n'avons pas pu gérer le cas deux chariot se succédant sur une même case. En effet, lors de l'avancée d'un chariot, la case où il était précédemment situé est effacé, et ce pour donner une illusion de mouvement : la représentation du chariot qui en suit en suite est alors effacé.

De plus, nous avons tardivement compris que la dimension spatio-temporelle entraîne un déroulement non-uniforme des différents chemins : ainsi, même s'il n'y a pas de collision au même index dans les chemins, le fait que des pauses de trois secondes soit effectuées à chaque tournant entraîne un décalage. Bien qu'il n'y ait pas de collision dans l'algorithme, le dessin laisse apparaître des collisions, due à la désynchronisation du dessin.

### **Instructions supplémentaires :**

- Par défaut, le programme démarre sur l'algorithme spatiotemporel, grâce à Form1. Trois chariots y sont instanciés : il ne reste plus qu'à cliquer sur "Go".
- Pour entrer des valeurs manuellement, il faut charger dans Program.cs le formulaire InitialiseForm.
- Pour utiliser l'algorithme du chemin le plus court sans la dimension spatiotemporelle, il suffit de charger Form2 dans Program.cs.
- Une interface complémentaire est disponible dans le dossier [Interface Chariots], plus user-friendly : l'utilisateur n'a plus qu'à cliquer sur la grille pour positionner ses chariots. Cependant, en raison de problèmes de formats d'écran n'ayant pas été complètement solutionné à temps, nous avons préféré opté pour une interface plus simple mais complètement fonctionnelle.

### **Conclusion et remarques**

Nous avons apprécié cet exercice : le sujet était original et intéressant, bien que complexe. Le temps de développement nous a aussi semblé trop bref : nous aurions d'une part aimé améliorer notre algorithme afin de régler les problèmes ciblés, et d'autre part intégrer des heuristiques à l'algorithme spatiotemporel (ne pas s'engager dans une allée si un chariot s'y trouve déjà par exemple).

La qualité du code s'en ressent aussi : si les fonctionnalités sont présentes et en majorité stables, nous n'avons pas eu le temps de "nettoyer" le code comme nous l'aurions voulu.

## **Partie II**

### **Introduction**

Dans cette partie, il s'agissait d'implémenter différentes techniques de machine learning. Pour chacune de ces techniques, une base d'exemples était proposée, afin d'entraîner l'algorithme étudié.

Dans le but de faciliter la compréhension et l'encapsulation des composants, nous avons usé de quelques concepts de programmation qu'il nous semble important de mentionner.

D'abord, nous avons constaté qu'aucune information d'entrée n'était dynamique, dans la mesure où l'on chargeait en mémoire seulement des exemples issus d'un fichier texte fourni et statique.

En conséquence, nous avons choisi d'implémenter un contrôleur statique, qui se chargerait de formater les données selon les besoins de l'algorithme en terme de paramètre d'entrée. Ainsi, chaque technique de machine learning implémentée possède son contrôleur qui relie sa base d'exemples textuels avec son propre apprentissage.

Cette façon de faire nous a ainsi permis d'épurer notre classe Vue qui exécute seulement le chargement de l'apprentissage avant d'afficher le résultat.

D'autre part, nous avons conçu un système de fabrique d'exemples à l'intérieur même du contrôleur. En effet, les algorithmes de machine learning prennent en paramètres un très grand nombre de données typées de différentes manières. Notre fabrique va alors s'occuper d'instancier tous les objets qui serviront d'exemples sous forme d'une collection, exploitable en l'état par l'algorithme.

Ces contrôleurs rassemblés constituent le design pattern Factory qui instancie différents types de Produits selon le fichier chargé.

Enfin, nous avons ajouté à nos contrôleurs des fonctionnalités de gestion des exemples. Les principales sont :

- La possibilité d'intercaler deux ensembles d'exemples ,
- Le mélange d'un ensemble d'entrées en conservant dans une collection séparée les sorties attendues dans le bon ordre,
- La normalisation d'un ensemble d'entrées sous forme de vecteurs à 2 dimensions.

D'autres fonctionnalités plus basiques sont présentes et ont facilité l'obtention de résultats probants.

Pour finir, nous avons remarqué que l'utilisation conjointe des principales fonctionnalités des contrôleurs nous permettaient d'obtenir des résultats très satisfaisants en optimisant le nombre d'itérations d'apprentissage.

### Question 1 : classification avec Perceptron 1 couche (1 point)

A l'issue des différentes phases d'itération d'ajustement des poids, les valeurs finales des  $w$  obtenues sont les suivantes :

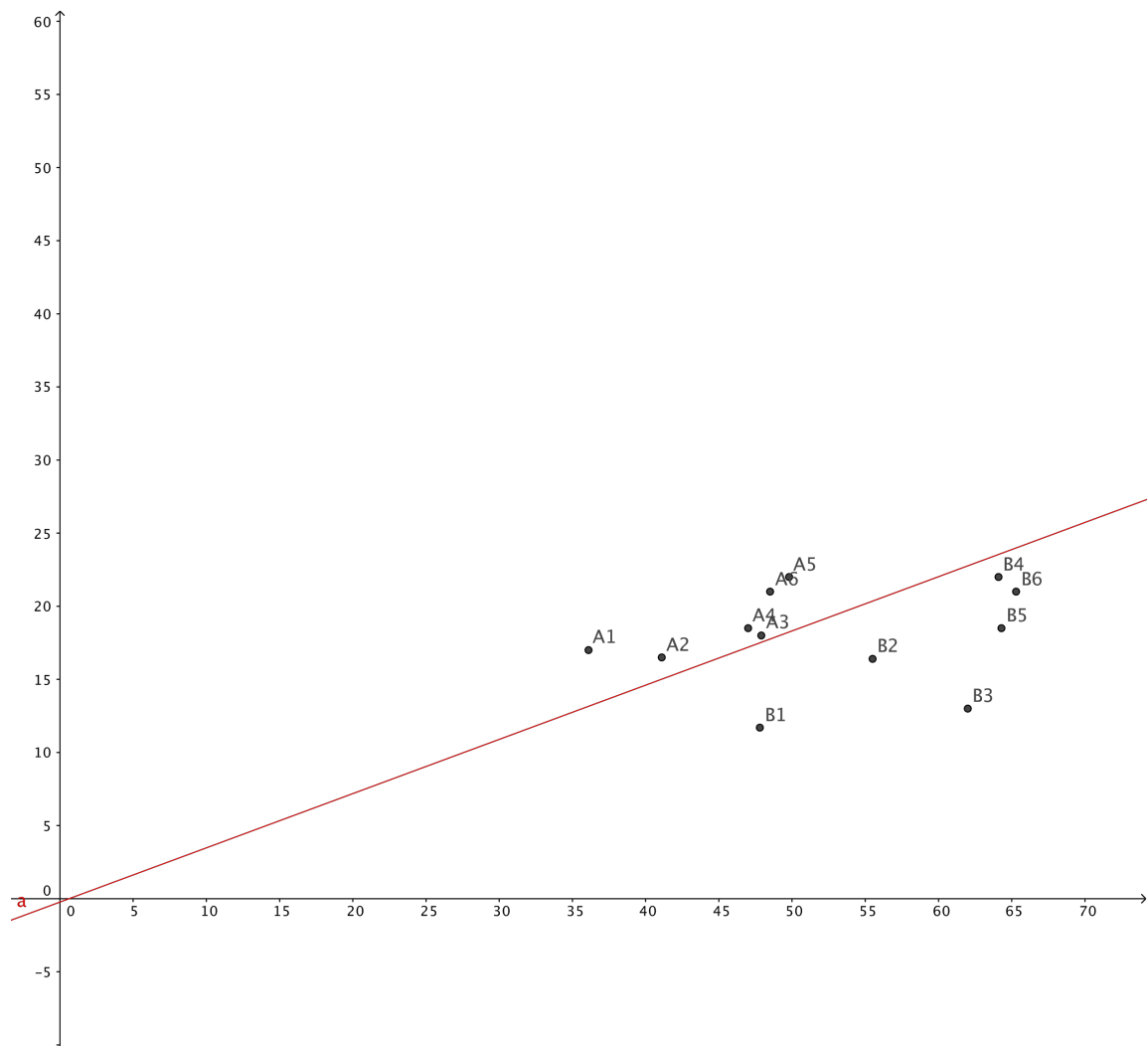
- $w1 = -14.5$  ;
- $w2 = 39.1$  ;
- $w3 = 9$ .

Le nombre d'erreurs commises par l'algorithme sur chaque itération est affiché ci-dessous :

```
1 = 1 | 1 = 0 | 0 = 1 | 1 = 0 | 0 = 1 | 1 = 0 | 0 = 1 | 1 = 0 | 0 = 1 | 0 = 0 | 0 = 1 | 1 = 0 | Nombre d'erreurs : 10
Nouvelle itération
0 = 1 | 1 = 0 | 0 = 1 | 1 = 0 | 0 = 1 | 1 = 0 | 0 = 1 | 0 = 0 | 0 = 1 | 1 = 0 | 0 = 1 | 1 = 0 | Nombre d'erreurs : 11
Nouvelle itération
0 = 1 | 1 = 0 | 0 = 1 | 0 = 0 | 1 = 1 | 0 = 0 | 1 = 1 | 0 = 0 | 1 = 1 | 0 = 0 | 1 = 1 | 0 = 0 | Nombre d'erreurs : 3
Nouvelle itération
1 = 1 | 0 = 0 | 1 = 1 | 0 = 0 | 1 = 1 | 0 = 0 | 1 = 1 | 0 = 0 | 1 = 1 | 0 = 0 | 1 = 1 | 0 = 0 | Nombre d'erreurs : 0
Nouvelle itération
Valeurs finales des poids :
w1 = -14,5
w2 = 39,1
w3 = 9
```

Le tracé de la droite définie par les différents poids a été réalisé sous GeoGebra.  
Son équation est donnée par la relation :

$$f(x,y) = -14.5x + 39.1y + 9 = 0$$




On repère bien de part et d'autre du séparateur linéaire les deux espèces animales A et B.

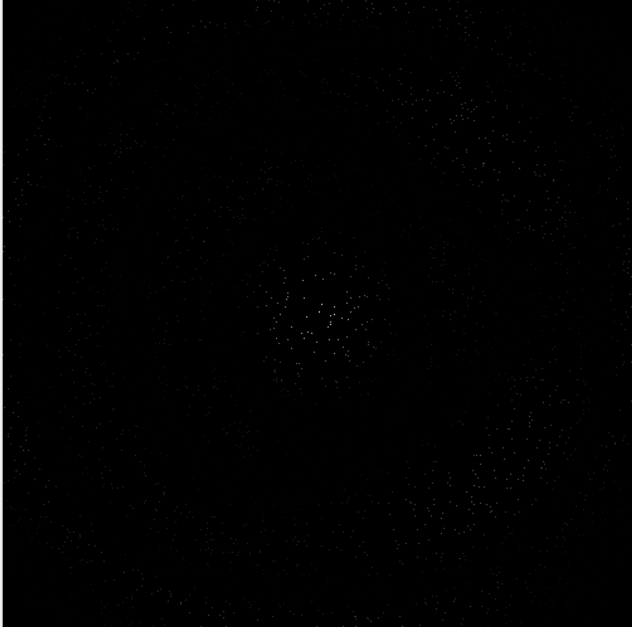
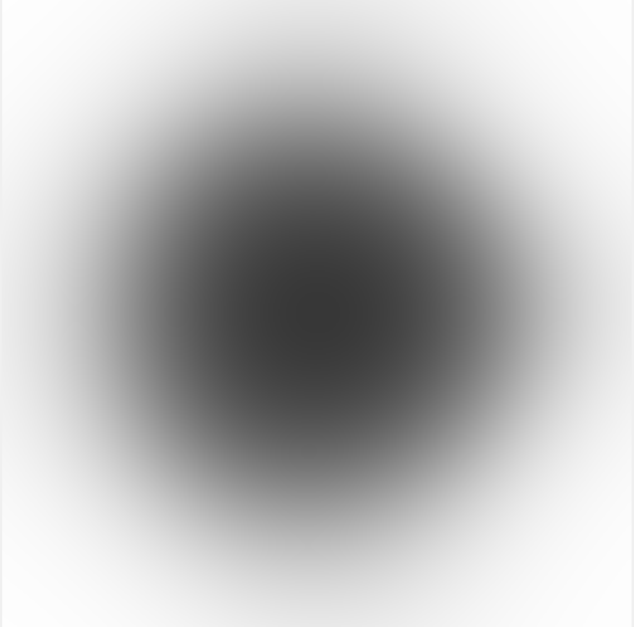
## Question 2 : régression avec un perceptron multi-couches (3 points)

Dans un premier temps, il convient de déterminer le “paramétrage” qui nous permettra de converger le plus rapidement possible vers le résultat souhaité. Nous allons donc dans un premier temps déterminer à tâtons le bon nombre d’itérations et de neurones par couche cachée, puis évaluer l’effet de la variation de ces paramètres sur la vitesse de convergence d’apprentissage.

A priori, il faut un nombre de :

- **100** itérations par apprentissage,
- **4** neurones par couche cachée,
- et un coefficient d’apprentissage égal à **2**.

nb entrées	<input type="text" value="3"/>	(y compris la constante)	Numéro couche	<input type="text" value="1"/>	Numéro neurone	<input type="text" value="0"/>	nb d'itérations pour chaque apprentissage	<input type="text" value="100"/>
nb couches	<input type="text" value="3"/>	(les entrées comptent pour 1 couche)					alpha (coefficient d'apprentissage)	<input type="text" value="2"/>
nb neurones par couche	<input type="text" value="4"/>	(par couche cachée)						
Obligatoire pour créer le réseau			Cliquez plusieurs fois pour converger					
<input type="button" value="Init réseau"/>			<input type="button" value="apprentissage"/>			<input type="button" value="Affiche info neurone"/>		
						Erreur max : 53,2133605286832		
						Taux d'erreur résiduel : 6,442621446058		



(1)

### Paramètres de sortie

Erreur max :	53,2133605286832
Taux d'erreur résiduel :	6,442621446058

Pour un nombre de **4** neurones par couche cachée et un coefficient d'apprentissage égal à **2**, le taux d'erreur résiduel obtenu à l'issue de **100** itérations est d'approximativement **6.4**.

Nous allons désormais étudier l'influence des paramètres “nombre de neurones par couche cachée” et “coefficient d'apprentissage” sur la vitesse de convergence de l'apprentissage.

### **A. Variation du paramètre “nombre de neurones par couche cachée”**

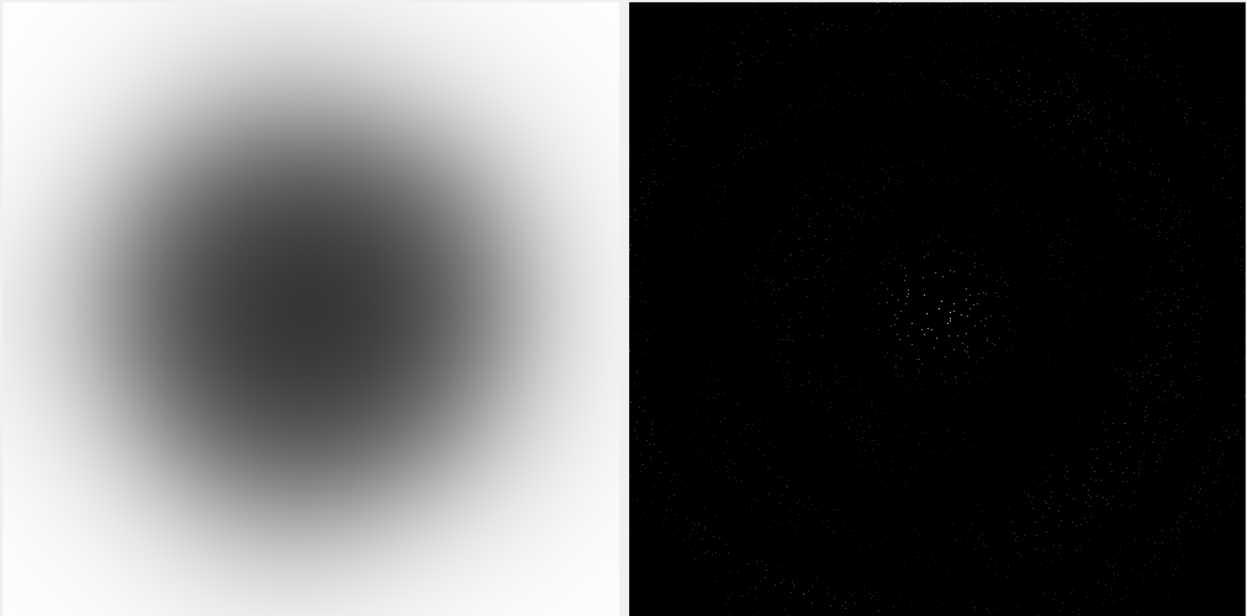
#### **Paramètres d'entrée**

*Nombre d'itérations par apprentissage : 100*

*Nombre de neurones par couche cachée : **10***

*Coefficient d'apprentissage : 2*

nb entrées	<input type="text" value="3"/>	(y compris la constante)	Numéro couche	<input type="text" value="1"/>	Numéro neurone	<input type="text" value="0"/>	nb d'itérations pour chaque apprentissage	<input type="text" value="100"/>
nb couches	<input type="text" value="3"/>	(les entrées comptent pour 1 couche)					alpha (coefficient d'apprentissage)	<input type="text" value="2"/>
nb neurones par couche	<input type="text" value="10"/>	(par couche cachée)						
Obligatoire pour créer le réseau		Cliquez plusieurs fois pour converger		Affiche info neurone				
<input type="button" value="Init réseau"/>		<input type="button" value="apprentissage"/>		<div></div>				
				Erreur max : 52.8896768226447				
				Taux d'erreur résiduel : 6.18660979246364				



(2)

#### **Paramètres de sortie**

Erreur max : 52.8896768226447

Taux d'erreur résiduel : 6.18660979246364

On remarque ici que le réseau apprend plus rapidement lorsque le nombre de neurones par couche cachée augmente. En effet, pour un même nombre d'itérations (100), le taux d'erreur résiduel obtenu après apprentissage diminue légèrement de quelques centièmes, passant de 6.4 à 6.2. Pour nous convaincre de l'effet positif de l'augmentation du nombre de neurones par couche cachée sur la vitesse de convergence d'apprentissage, nous pouvons choisir un nombre d'itérations, un coefficient d'apprentissage ainsi qu'un nombre de neurones par couche cachée plus faibles, puis comparer les résultats obtenus :

*Nombre d'itérations par apprentissage : 5*

*Nombre de neurones par couche cachée : 4*

*Coefficient d'apprentissage : 1*

nb entrées	<input type="text" value="3"/>	(y compris la constante)	Numéro couche	<input type="text" value="1"/>	Numéro neurone	<input type="text" value="0"/>	nb d'itérations pour chaque apprentissage	<input type="text" value="5"/>
nb couches	<input type="text" value="3"/>	(les entrées comptent pour 1 couche)					alpha (coefficient d'apprentissage)	<input type="text" value="1"/>
nb neurones par couche	<input type="text" value="4"/>	(par couche cachée)						

Obligatoire pour créer le réseau

Cliquez plusieurs fois pour converger

Erreur max : 138,34379781859

Taux d'erreur résiduel : 48,4917835559327

(3)

### Paramètres de sortie

Erreur max :	138,34379781859
Taux d'erreur résiduel :	48,4917835559327

Nombre d'itérations par apprentissage : 5  
 Nombre de neurones par couche cachée : 6  
 Coefficient d'apprentissage : 1

nb entrées	<input type="text" value="3"/>	(y compris la constante)	Numéro couche	<input type="text" value="1"/>	Numéro neurone	<input type="text" value="0"/>	nb d'itérations pour chaque apprentissage	<input type="text" value="5"/>
nb couches	<input type="text" value="3"/>	(les entrées comptent pour 1 couche)						
nb neurones par couche	<input type="text" value="6"/>	(par couche cachée)						

Obligatoire pour créer le réseau

Cliquez plusieurs fois pour converger

alpha (coefficient d'apprentissage)

Erreur max : 76,1829898632673

Taux d'erreur résiduel : 20,7299844729899

(4)

### Paramètres de sortie

Erreur max :	76,1829898632673
Taux d'erreur résiduel :	20,7299844729899

En faisant varier le nombre de neurones par couche cachée de 4 à 6, on note que le taux d'erreur résiduel bascule de 48.5 à 20.7, soit diminue plus de moitié. Qualitativement, on note que lorsque le nombre de neurones par couche cachée augmente, l'image affichée à droite (représentation de l'erreur en valeur absolue sous forme d'un niveau de gris) tend d'autant plus vers un noir uniforme, ce qui vient confirmer les observations quantitatives faites.



## **B. Variation du paramètre “coefficient d’apprentissage”**

Sur le même principe, nous allons ici garder un nombre de 5 itérations par apprentissage pour pouvoir mieux apprécier l’effet induit par la variation de ce deuxième paramètre. Notre référentiel sera celui dont l’image 3 est issue, défini par un nombre de 5 itérations par apprentissage, 4 neurones par couche cachée et un coefficient d’apprentissage égal à 1. Voyons maintenant l’effet produit par une augmentation du coefficient d’apprentissage, passant de 1 à 9 :

*Nombre d’itérations par apprentissage : 5*

*Nombre de neurones par couche cachée : 4*

*Coefficient d’apprentissage : 9*

nb entrées : 3 (y compris la constante)  
nb couches : 3 (les entrées comptent pour 1 couche)  
nb neurones par couche : 4 (par couche cachée)

Numéro couche : 1  
Numéro neurone : 0

nb d'itérations pour chaque apprentissage : 5

alpha (coefficient d'apprentissage) : 9

Obligatoire pour créer le réseau : Init réseau

Cliquez plusieurs fois pour converger : apprentissage

Affiche info neurone

Erreur max : 40,9165794142222  
Taux d'erreur résiduel : 12,454473299179

(5)

## **Paramètres de sortie**

Erreur max : 40,9165794142222  
Taux d'erreur résiduel : 12,454473299179

Même chose ici, la vitesse de convergence liée à l’apprentissage est d’autant plus importante lorsque le coefficient d’apprentissage l’est aussi. Pour un nombre identique de neurones par

couche cachée (soit 4), le taux d'erreur résiduel obtenu bascule de 48.5 à 12.5, soit une diminution de 36 points, pour une erreur maximale là encore beaucoup plus faible.

Il semblerait, compte tenu des résultats obtenus après modification de ces deux paramètres, que le coefficient d'apprentissage soit celui qui joue le plus fortement sur la vitesse de convergence d'apprentissage du perceptron.

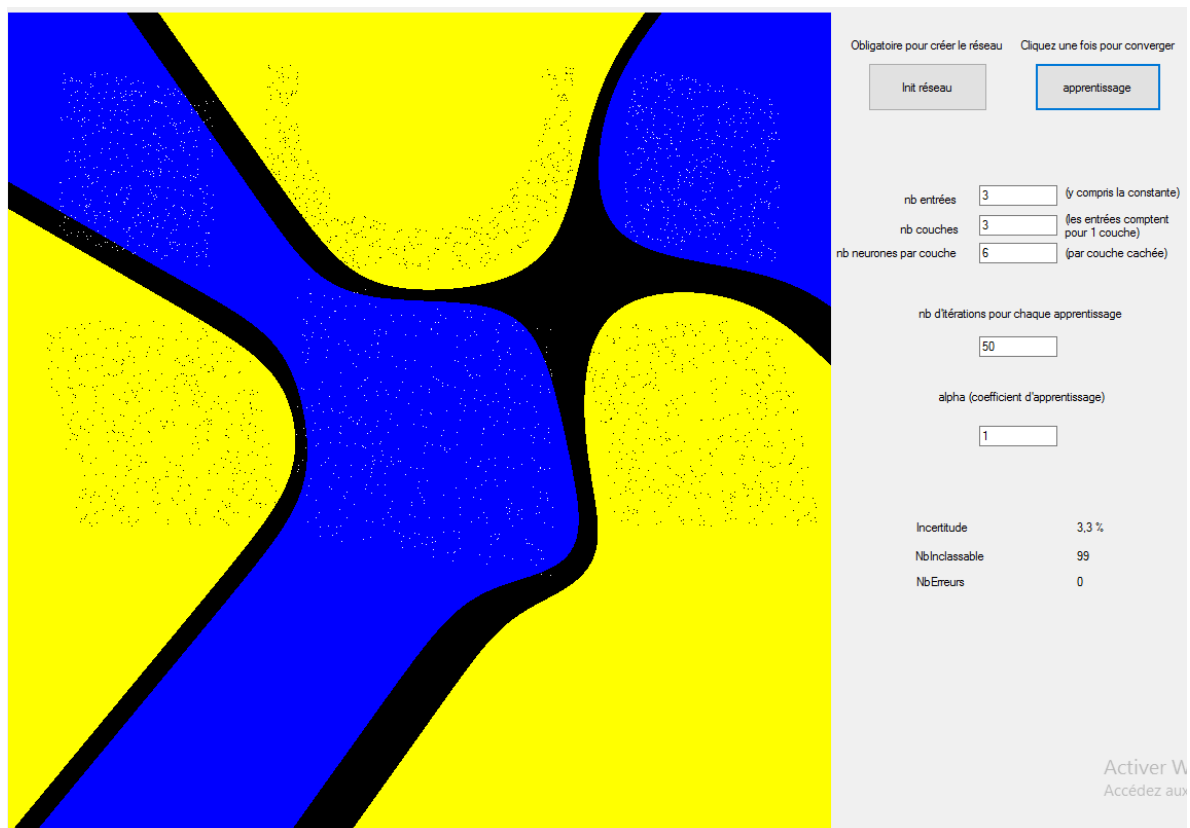
Il semblerait que la fonction mathématique approximativement encodée par le perceptron  $z = f(x,y)$  tende vers 0 lorsque  $x$  et  $y$  tendent vers  $(tailleImage/2)$ .

### Question 3 : classification

#### 3.1. Apprentissage supervisé

Pour obtenir le meilleur résultat en un minimum de temps, nous avons décidé d'instancier les paramètres suivants comme tel :

- nombre d'itérations par apprentissage : 50
- nombre de neurones par couche cachée : 6
- coefficient d'apprentissage : 1



#### Paramètres de sortie

Incertitude	3,3 %
NbInclassable	99
NbErreurs	0

A l'issue de ce premier apprentissage supervisé sur 50 itérations, la classification des points selon leur classe d'appartenance apparaît plutôt bonne, avec d'un côté les points noirs (située dans les zones jaunes) et de l'autre les points blancs (sur fond bleu). Le taux d'incertitude est relativement faible (3,3%), ce qui vient conforter les observations faites.

L'influence des paramètres “nombre de neurones par couche cachée” et “coefficient d'apprentissage” va maintenant être étudiée.

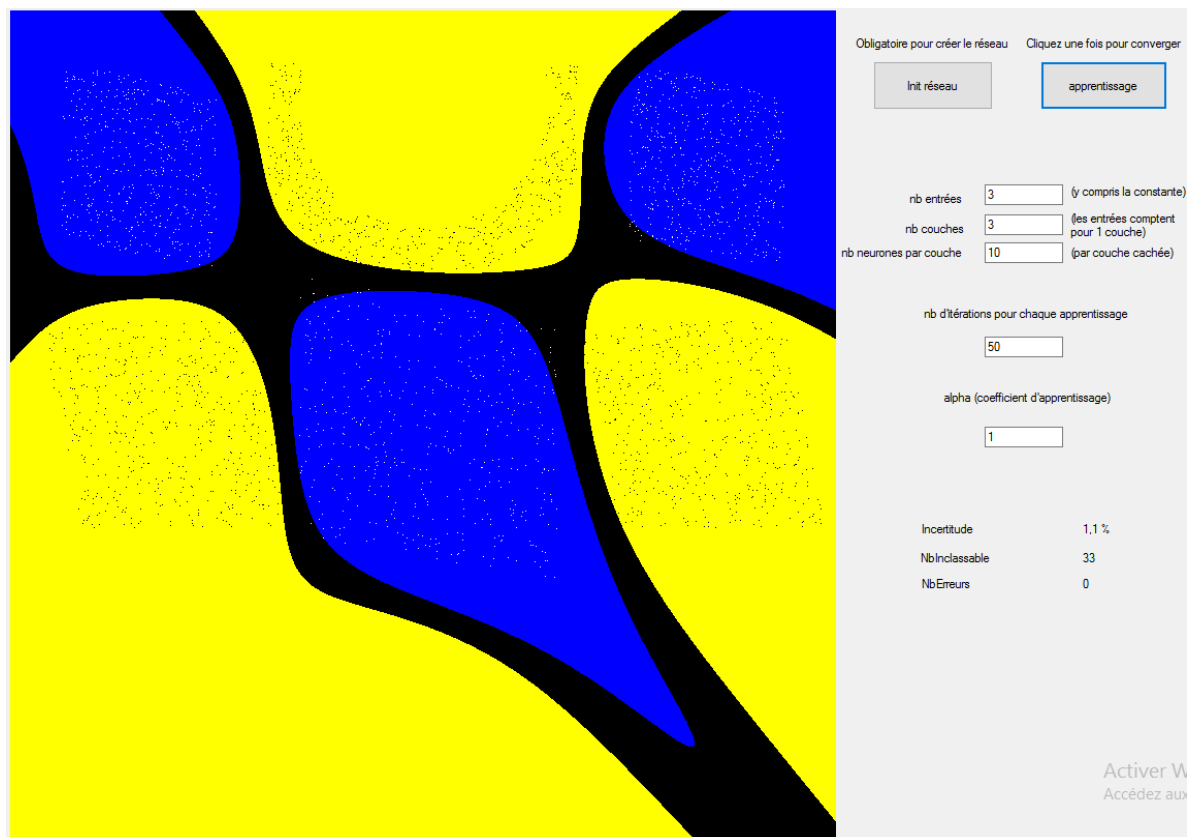
## **A. Variation du paramètre “nombre de neurones par couche cachée”**

### **Paramètres d'entrée**

*Nombre d'itérations par apprentissage : 50*

*Nombre de neurones par couche cachée : 10*

*Coefficient d'apprentissage : 1*



### **Paramètres de sortie**

Incertitude	1,1 %
NbInclassable	33
NbErreurs	0

En faisant passer le nombre de neurones par couche cachée de 6 à 10, on remarque que la qualité des résultats obtenus est bien meilleure, ce qui se traduit tant qualitativement (bonne classification des points dans les différentes zones) que quantitativement, avec un taux d'incertitude plus faible (1,1% contre 3,3%). Le nombre d'inclassables a là aussi diminué, de 63 points.

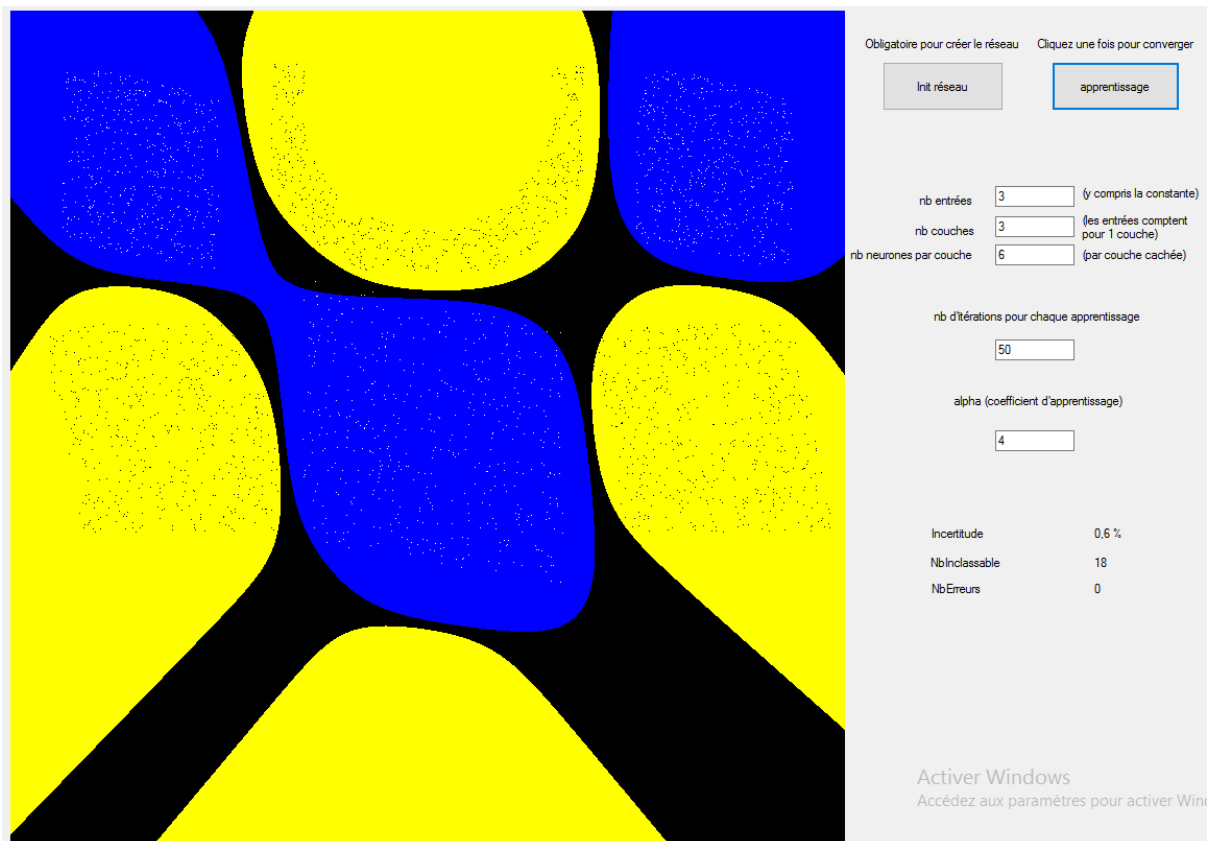
## **B. Variation du paramètre “nombre de neurones par couche cachée”**

### **Paramètres d'entrée**

*Nombre d'itérations par apprentissage : 50*

*Nombre de neurones par couche cachée : 6*

*Coefficient d'apprentissage : 4*



### **Paramètres de sortie**

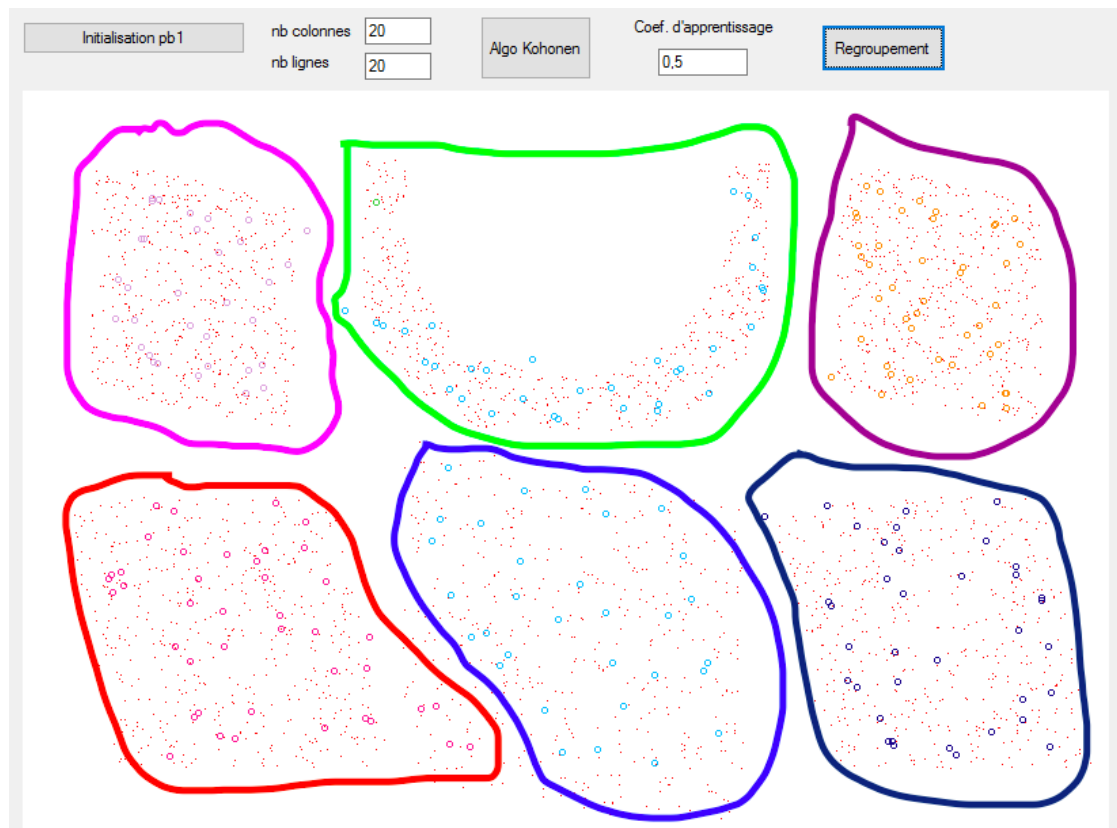
Incertitude	0.6 %
NbInclassable	18
NbErreurs	0

L'augmentation du coefficient d'apprentissage (+3) a un effet positif sur les résultats obtenus: la classification apparaît encore bien meilleure que dans les deux cas précédents, le taux d'incertitude étant désormais inférieur à 1%.

### 3.2. Apprentissage non supervisé

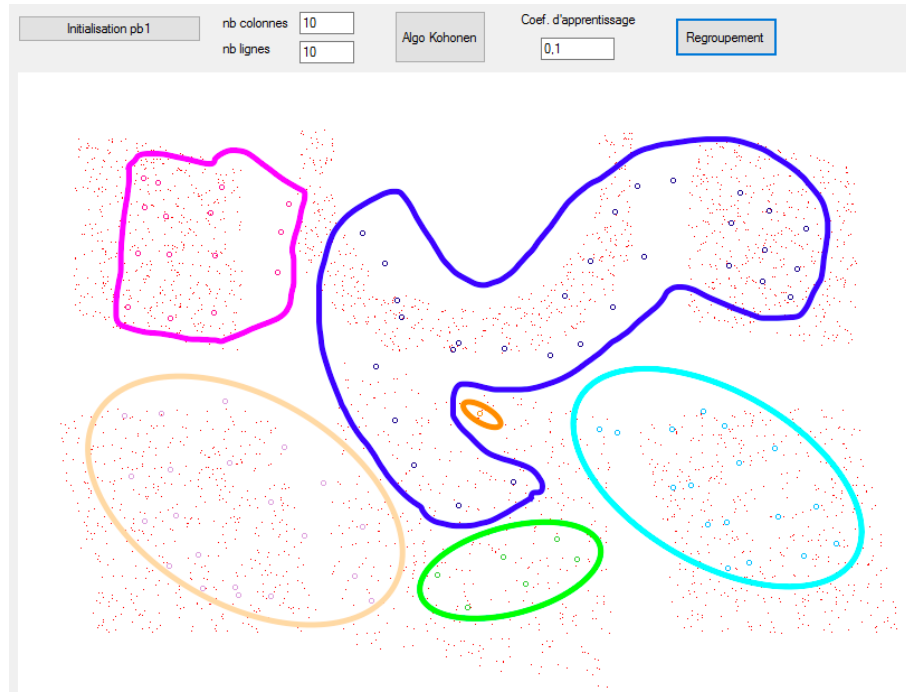
Pour l'apprentissage non supervisé, l'algorithme de Kohonen a été implémenté sur la base d'une liste d'Observations, intercalée et mélangée. Nous avons mis au point 6 classes de différentes couleurs selon les 6 zones de points.

Le meilleur résultat obtenu comportait une carte de 20x20 neurones, avec un coefficient d'apprentissage de 0,5 :

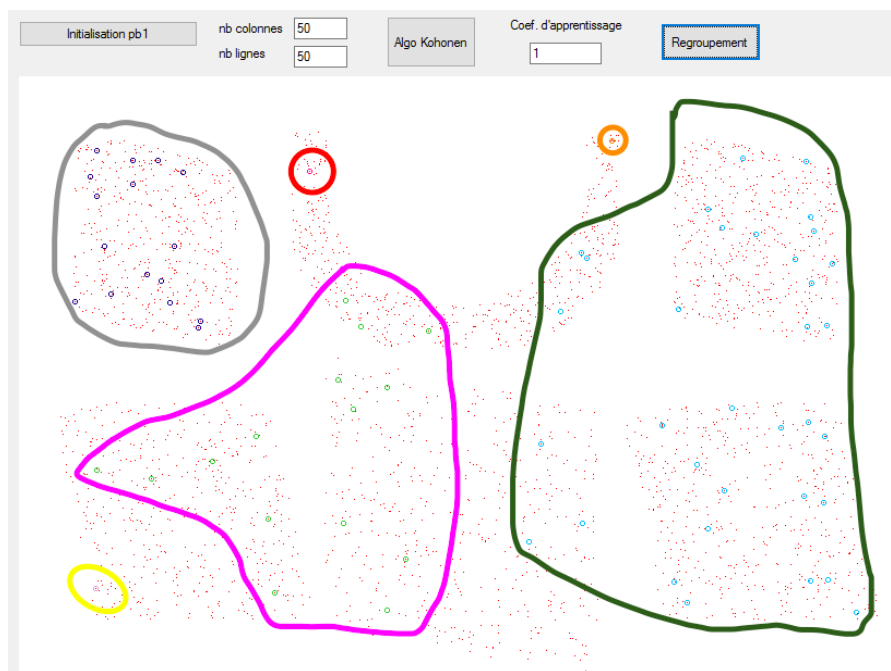


## Variation du nombre de neurones de la carte

En réduisant le nombre de neurones de la carte, nous avons constaté que l'algorithme convergait moins bien.

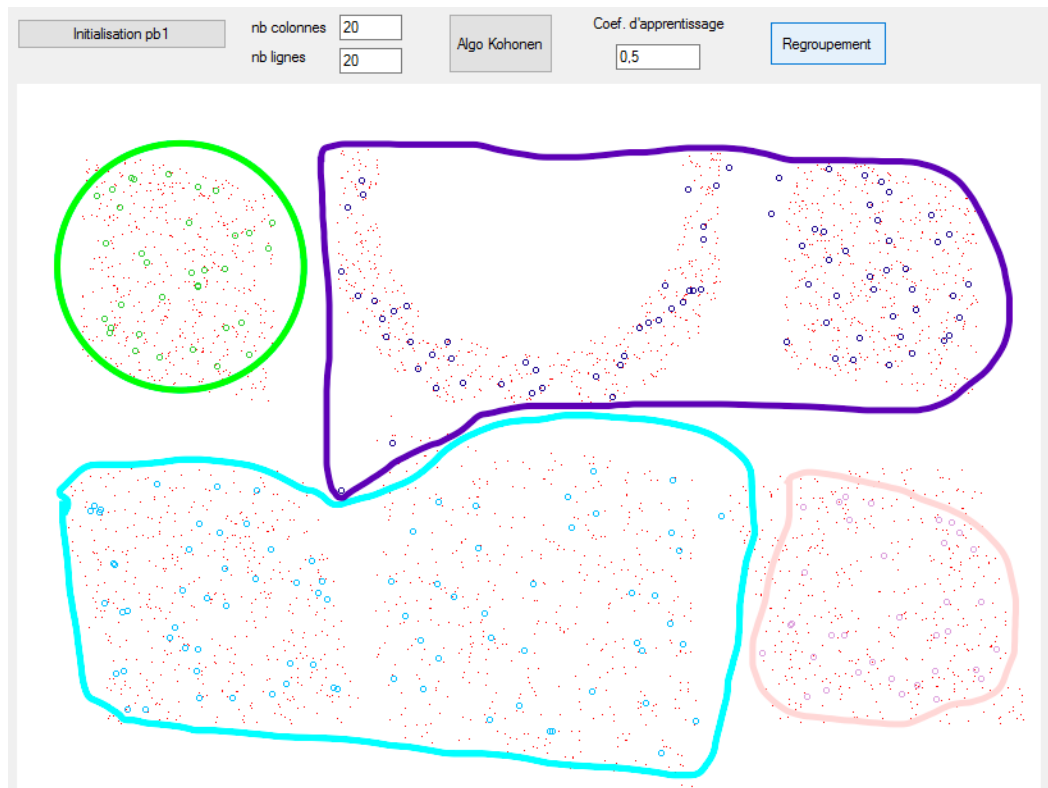


En faisant passer le coefficient d'apprentissage à 1, le temps de calcul s'est avéré trop long pour des résultats bien médiocres :



## Variation du nombre de classes pour le regroupement

Nous avons fait passer le nombre de classes de regroupement de 6 à 4.



Les valeurs optimales sélectionnées pour les 6 classes sont aussi les meilleures valeurs pour les 4 classes.

En définitive, il semblerait que ce soit le paramètre “nombre de neurones sur la carte” qui augmente le plus largement la vitesse de convergence. Ceci dit, une configuration de 20x20 neurones est acceptable, et le temps de calcul par clic sur “Apprentissage” est imperceptible.

### Gestion du projet : répartition des tâches

Partie I	Question/Tâche	Elève(s)
	<i>Question 1</i>	Hadrien, Rodolphe, Solène
	<i>Question 2</i>	Rodolphe, Solène
	<i>Question 3</i>	Rodolphe
Partie II	<i>Question 1</i>	Hadrien, Rodolphe, Solène
	<i>Question 2</i>	Hadrien et Solène
	<i>Question 3 - apprentissage supervisé</i>	Hadrien et Solène
	<i>Question 3 - apprentissage non supervisé</i>	Hadrien