CFPT-INFORMATIQUE

Travaux de Diplômes 2017

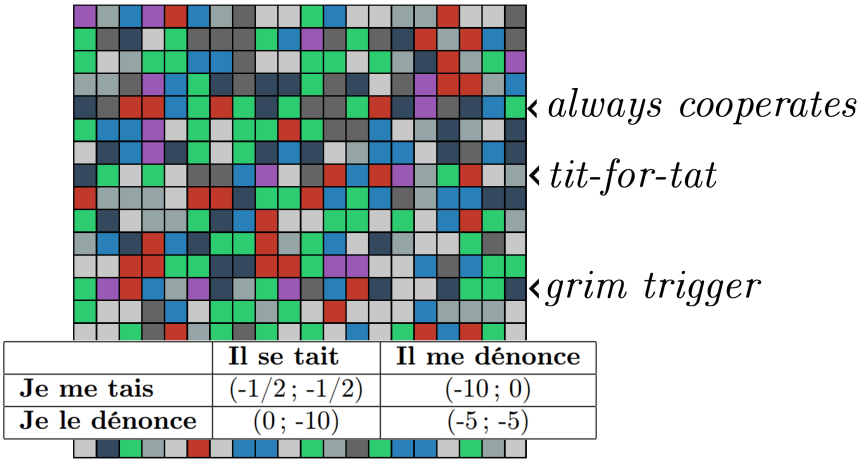
Dilemme du Prisonnier

Automate Cellulaire

Julien SEEMULLER

Supervisé par :

Mme. Terrier



	Il se tait	**Il me dénonce**
Je me tais	(-1/2 ; -1/2)	(-10 ; 0)
Je le dénonce	(0 ; -10)	(-5 ; -5)

T.IS-E2B

12 juin 2017

1 Abstract

The purpose of this project is to create a testing environment for the prisoner's dilemma. The prisoner's dilemma is a well known example in the field of game theory, where two players compete in a *zero-sum game* (a game where one player's gains results in losses for the other player).

In this application, a cellular automaton is modeled after the prisoner's dilemma. A cellular automaton is a grid of colored cells that evolves over time. Each cell has a strategy and plays with each of it's nearest neighbors (one cell apart from our current cell, similar to how the king moves in the game of chess). The user can step forward in time at the press of a button, while data are plotted live on different charts. Girds can also be randomly generated and the payoff of each outcome of the game can be adjusted by means of a payoff matrix.

The prisoner's dilemma cellular automaton is a flexible approach to testing an IPD (iterated prisoner's dilemma) strategy. Strategies can be easily implemented, tested against each other and analysed with charts.

2 Résumé

Le but de ce projet est de réaliser un environnement de test pour le dilemme du prisonnier. Le dilemme du prisonnier est un exemple connu dans le domaine de la théorie des jeux, ou deux joueurs s'opposent dans un jeu a somme nulle (un jeu ou les gains d'un des joueurs sont égaux aux pertes de l'autre joueur).

Dans cette application, un automate cellulaire est conçu sur la base du dilemme du prisonnier. Un automate cellulaire est une grille de cellules colorées évoluant au fil du temps. Chaque cellule possède une stratégie et joue avec ses voisins les plus proches (cellules adjacentes à la cellules actuelle, similaire aux mouvements possible d'un roi dans le jeu d'échecs). L'utilisateur peut faire progresser le jeu en appuyant sur un bouton, et les données résultantes sont affichées en temps réel sur des graphiques. Des grilles de cellules peuvent aussi être générées aléatoirement, et la matrice représentant les gains des cellules peut être ajustée.

L'automate cellulaire du dilemme du prisonnier est une solution flexible pour tester des stratégies du DPR (dilemme du prisonnier répété). Diverses stratégies peuvent être ajoutées facilement, testées les unes contre les autres et analysées a l'aide de graphiques.

Table des matières

3 Introduction

Ce projet à été réalisé dans le cadre des travaux de diplômes de l'année 2016-2017 du **C**entre de **F**ormation **P**rofessionnelle **T**echnique en **I**nformatique dans l'optique d'obtenir un diplôme de Technicien ES.

Durant l'année, nous avons effectués plusieurs travaux sur les automates cellulaires, un sujet auquel je porte un grand intérêt. En lisant différents articles sur internet, il m'est venu l'idée d'un automate cellulaire basé sur le dilemme du prisonnier. Cependant, la majorité de ces articles parviennent de personnes ayant un bagage scientifique en physique, et les démonstrations illustrées dans ces derniers sont réalisés dans des langages axés mathématiques tels que "R".

J'ai comme but de réaliser cet automate cellulaire entièrement en C# et d'utiliser une bibliothèque permettant d'afficher mes résultats dans un format clair et intuitif à l'utilisateur.

J'ai également comme but d'appliquer les concepts souvent survolés lors des travaux de diplômes tels que le développement piloté par les tests (*test driven development* en anglais) ou encore les patrons de conceptions (*design patterns* en anglais). Ces différents concepts assurent une architecture plus cohérente, et facilite la compréhension pour les personnes extérieures au projet.

4 Cahier des charges

4.1 Sujet

Automate cellulaire (voir *Conway's Game of Life* [1]) basé sur le dilemme du prisonnier répété (voir *Iterated Prisoner's Dilemma* [2]) et permettant de le simuler.

4.2 Descriptions

Le projet étant basé sur deux concepts peu courants, il est nécessaire de les détailler.

Automate cellulaire :

Un automate cellulaire est un modèle constitué d'une grille de cellule changeant d'état à chaque temps $t+1$. Une règle est appliquée à toutes les cellules, habituellement basée sur l'état des voisins de chaque cellule, et permet de faire "évoluer" la grille. L'automate cellulaire le plus connu est probablement *Game of Life* imaginé par John Horton Conway en 1970.

Dilemme du prisonnier répété :

Le dilemme du prisonnier répété est une variante du dilemme du prisonnier. Dans ce jeu, des personnes jouent plusieurs fois au dilemme du prisonnier.

Dans le dilemme du prisonnier, deux prisonniers ayant commis un crime mineur sont enfermés dans deux cellules différentes, afin de les empêcher de communiquer. Le policier soupçonne les deux accusés d'avoir commis auparavant un crime plus important et souhaite obtenir des aveux concernant ce dernier. Il se présente donc et discute avec chaque prisonnier séparément en leur offrant à chacun deux choix :

- Dénoncer l'autre prisonnier (trahison)
- Se taire (coopération)

Il présente donc les résultats des choix suivants :

- Si l'un des deux prisonniers dénonce l'autre, il est remis en liberté alors que le second obtient la peine maximale (10 ans)
- Si les deux se dénoncent entre eux, ils seront condamnés à une peine plus légère (5 ans)
- Si les deux refusent de dénoncer l'autre, la peine sera minimale (6 mois), faute d'éléments à charge.

Chaque prisonnier fait donc une *"Matrice des Gains"* [3] pour résoudre ce problème :

	Il se tait	**Il me dénonce**
Je me tais	(-1/2 ; -1/2)	(-10 ; 0)
Je le dénonce	(0 ; -10)	(-5 ; -5)

Chaque prisonnier *devrait* donc comprendre que le choix logique sur une seule itération est de coopérer avec l'autre prisonnier.

4.3 But

Le but du projet est donc de fusionner ces deux concepts et de créer un automate cellulaire permettant de visualiser le dilemme du prisonnier répété. Chaque cellule jouerait une partie du dilemme simultanément avec chacun de ses voisins. Chaque cellule peut adopter une stratégie permettant d'optimiser ses gains. Voici quelques exemples de stratégies :

Random (RAND) : Fait des actions aléatoires, trahit ou coopère avec 50% de chance.
Always Defect (AD) : Trahit avec 100% de chance.
Always Cooperate (AC) : Coopère avec 100% de chance
Grim Trigger (GRIM) : Stratégie "AC", mais change sa stratégie vers "AD" après trahison.
etcetera... [4]

Beaucoup de stratégies peuvent être implémentées pour rendre le jeux intéressant à étudier. Pour cela, des graphiques seront implémentés permettant de récupérer et d'observer les résultats de l'application en temps réel. Les cellules du plateau seront aussi colorées selon leur stratégie ou encore l'historique de leur actions (ex : tendance à trahir → rouge et tendance à coopérer → vert).

Voici en exemple, le dilemme du prisonnier sous forme d'automate cellulaire :

A	*B*	*A*	*B*	*A*	*B*
B	*A*	*B*	*A*	*B*	*A*
A	*B*	*A*	*B*	*A*	*B*
B	*A*	*B*	*A*	*B*	*A*
A	*B*	*A*	*B*	*A*	*B*
B	*A*	*B*	*A*	*B*	*A*

$T=1$

θ						p					
0	0	0	0	0	0	16	16	16	16	16	16
0	0	0	0	0	0	16	16	**13**	16	16	16
0	0	π	0	0	0	16	**13**	**20**	**13**	16	16
0	0	0	0	0	0	16	16	**13**	16	16	16
0	0	0	0	0	0	16	16	16	16	16	16
0	0	0	0	0	0	16	16	16	16	16	16

$T=2$

θ						p					
0	0	0	0	0	0	16	**13**	16	**13**	16	16
0	π	0	π	0	0	**13**	**20**	**7**	**20**	**13**	16
0	0	π	0	0	0	16	**7**	**20**	**7**	16	16
0	π	0	π	0	0	**13**	**20**	**7**	**20**	**13**	16
0	0	0	0	0	0	16	**13**	16	**13**	16	16
0	0	0	0	0	0	16	16	16	16	16	16

FIGURE 1 – Automate cellulaire du dilemme du prisonnier
source [5]

4.4 Spécifications

Le projet défini dans le cadre suivant :

Automate cellulaire paramétrable :
- Nombre de cellules paramétrables.
- Stratégies utilisées et proportions de ces dernières sur le plateau paramétrables.
- Matrice des gains [3] paramétrable.

Exploitation des résultats :
- Cellules colorées selon leur stratégie ou historique d'actions.
- Divers graphiques (ex : nombre de cellules traîtres par générations)
- Possibilité d'utilisation de LiveCharts [6]

4.5 Environnement

Le projet prendra place dans l'environnement suivant :
- Ordinateur sous `Windows 7`
- Environnement de développement adapté pour `C#`.

4.6 Livrables

Les documents suivant seront remis à la fin du projet :

- Journal de bord (PDF).
- Rapport technique (PDF).
- Fichier ZIP contenant les sources.

4.7 Reddition

Voici les dates importantes du projet :

22 Janvier 2017 :	Reddition du cahier des charges.
5 Avril 2017 :	Début du travail
à définir :	Rendu du poster.
à définir :	Reddition intermédiaire de la documentation.
12 Juin 2017 :	Reddition finale du projet.
19-20 Juin 2017 :	Présentation orale du projet.

5 Planification provisionnelle

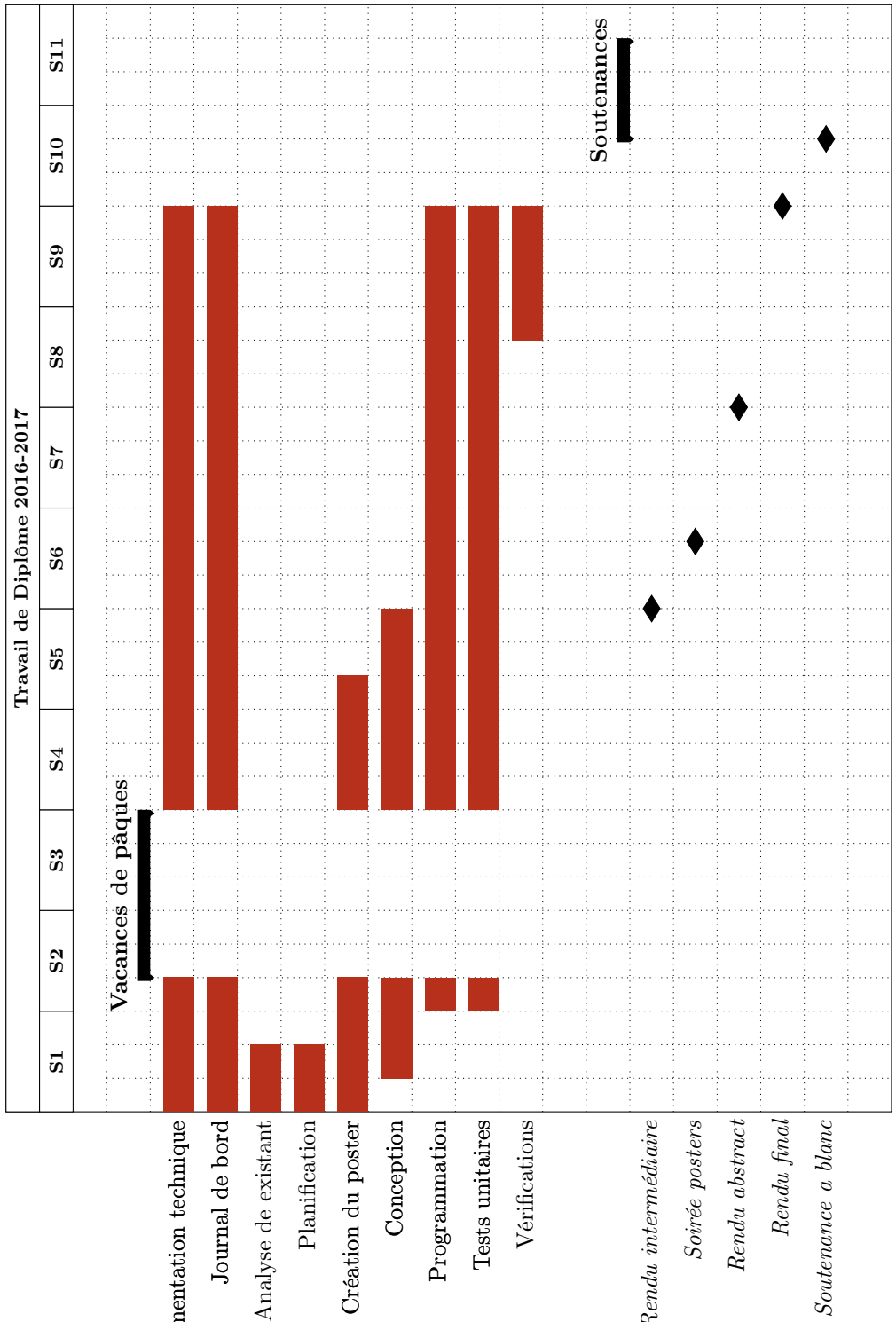


FIGURE 2 – Diagramme de Gantt

6 Analyse de l'existant

Il est nécessaire d'analyser et de comparer différents travaux avant de commencer le développement de notre application. Pour effectuer cette analyse, trois concepts d'automates cellulaires basés sur le dilemme du prisonnier ont étés sélectionnés :

— "*A quantum prisoner's dilemma cellular automaton*" de M. Ramón Alonso-Sanz. [7]
— "*Prisoner's dilemma in one-dimensional cellular automata*" de M. Marcelo Alves Pereira [8]
— "*The prisoner's dilemma and the game of life*" de Mme. Katarzyna Zbieć [9]

6.1 Projet de M. Ramón Alonso-Sanz

Le projet de M. Ramón Alonso-Sanz intitulé "*A quantum prisoner's dilemma cellular automaton*" reprends le dilemme du prisonnier de base, mais y ajoute quelques subtilités :

Le plateau est structuré sous la forme d'un échiquier, chaque cellule a donc quatre alliés et quatre rivaux, comparé à la forme habituelle, qui est d'utiliser les huit voisins de chaque cellules (similaire aux mouvements d'un roi dans le jeu des échecs). Les cellules possèdent des stratégies dites "quantiques" et adaptent aussi leurs stratégies à celle de leurs voisins. Les voisins ayant les meilleures performances sont imités par les autres cellules à l'aide d'une méthode nommé *imitation-of-the best*. Chaque cellule joue aussi avec elle-même en plus de ses rivaux. Ceci permet de prendre en compte ses propres résultats en faisant la moyenne des résultats obtenus entre les parties.

Un mécanisme de mémoire est aussi présent dans le programme de M.Ramon Alonso-Sanz. Ce dernier est de type "Markovien" (voir "chaînes de markov"), un historique complet n'est pas stocké mais les résultats et les choix précédents affectent les choix futurs de chaque cellule.

On compare aussi les stratégies dites "quantiques" aux stratégies classiques pour évaluer l'efficacité de ces dernières. Voici à quoi ressemble le projet de M. Ramón Alonso-Sanz :

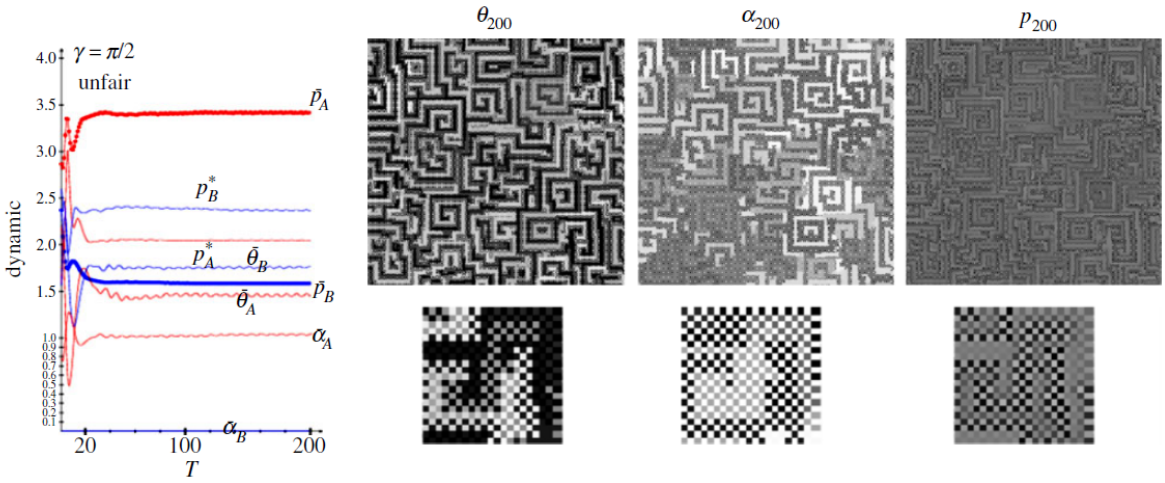


FIGURE 3 – Comparaison de stratégies quantiques (p_A) et classiques (p_B)

6.2 Projet de M. Marcelo Alves Pereira

Le projet de M. Marcelo Alves Pereira possède quelques différences avec un automate cellulaire du dilemme du prisonnier standard. En effet, M. Marcelo Alves Pereira allègue que la majorité des automates cellulaires basés sur le dilemme du prisonnier utilisent des structures trop complexes et suggère ainsi une approche plus simple. Ce dernier modélise le dilemme du prisonnier sous la forme d'un treillis à une dimension (tableau à une dimension), mais sa structure comporte quelques subtilités.

La première subtilité est le fait d'empiler ces tableaux à une dimension pour former un tableau en deux dimensions ou chaque position Y du tableau correspond à un temps T d'une partie. Ce système permet d'avoir en *tout temps* un historique complet et visible de la partie actuelle du dilemme du prisonnier. Voici un schéma représentant le fonctionnement de cette approche :

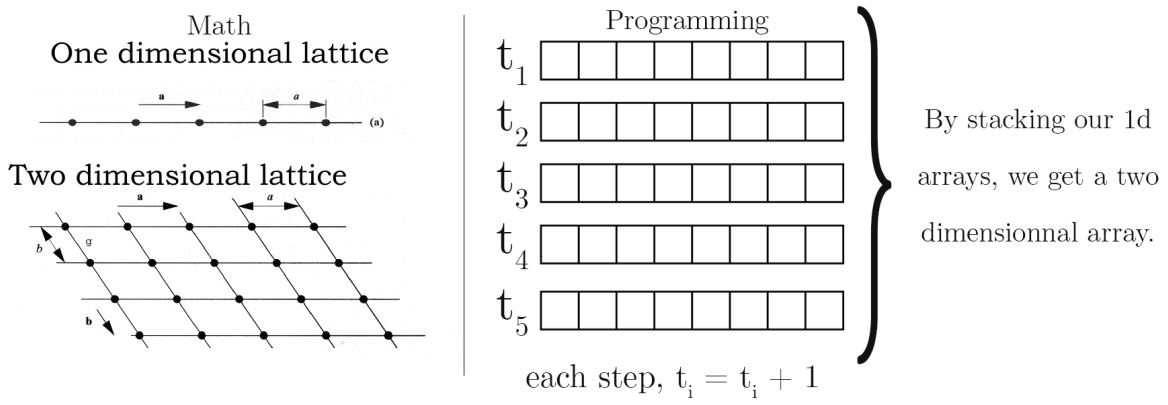


FIGURE 4 – Utiliser la deuxième dimension d'un tableau pour garder un "historique"

La deuxième subtilité est d'utiliser un nombre variable de voisins. En effet, le tableau étant sur un axe unique, on peut représenter le nombre de voisins de chaque cellule simplement par un chiffre X étant pair. Par exemple, pour 6 voisins par cellule, on considère les trois cellules à gauche et a droite de notre cellule actuelle comme nos voisins.

La troisième subtilité est d'utiliser le principe de *self-interaction* (ou "interaction avec soi" en français). Le principe est de "jouer" avec soi-même (la cellule actuelle) pour compenser un manque de joueurs quand le nombre de voisins n'est pas pair (par exemple, sur les bords de la matrice).

Malgré ces subtilités, ce système n'est pas parfait. Les cellules de ce système n'utilisent qu'une seule stratégie : celle du "*tit-for-tat*" (TFT) [4]. Avec cette stratégie, les cellules commencent dans un état aléatoire et copient la stratégie du voisin ayant obtenu le meilleur score. Ainsi, le jeu devient prévisible ; les cellules essaient de maximiser leurs gains de manière "égoïste" et des grappes de cellules trahissant leurs voisins se forment rapidement. Ce système n'est pas une mauvaise représentation du dilemme du prisonnier mais il serait intéressant d'ajouter plus de variations a ce dernier.

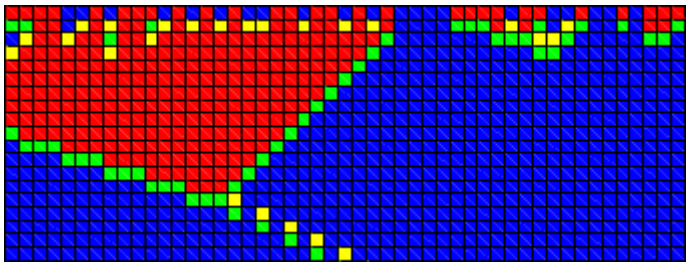


FIGURE 5 – Grappe de cellules "traîtres"

6.3 Projet de Mme. Katarzyna Zbieć

L'approche de Mme. Katarzyna Zbieć est différente des deux projets précédents. Elle vise à combiner le jeu de la vie de Conway et le dilemme du prisonnier. Les différents principes du jeu de la vie (cellules, états, plateau, etc...) et du dilemme du prisonnier (stratégies, matrice de gains, etc...) sont expliqués en détails et par la suite comparés.

Malheureusement, aucun exemple graphique n'est fourni avec le document. Cependant, ce projet reste le plus proche à celui qui sera développé lors de ce travail de diplôme.

Voici un tableau tiré du document de Mme. Katarzyna Zbieć ainsi que sa traduction française. Ces derniers font ressortir les ressemblances entre la structure du jeu de la vie et celle du dilemme du prisonnier :

the Prisoner's Dilemma	the Game of Life
the future of any player depends on the strategy of his/her neighbours	the future of any cell is determined by the state of its neighbours
the players are changing their own strategies in the way determined by the strategies of their enemies	the cells are changing colours in the way determined by the colours of their neighbours
the player can choose one of the two options: to cooperate or to defect	the cell has one of two states: live or dead
strategies	rules

FIGURE 6 – Comparaison entre le jeu de la vie et le dilemme du prisonnier

Le Dilemme du Prisonnier	Le Jeu de la Vie
Le futur de chaque joueur dépend de la stratégie de ses voisins	Le futur de chaque cellule est déterminé par l'état de ses voisins
Les joueurs changent leurs stratégies en se basant sur la stratégie de leurs ennemis	Les cellules changent de couleur en fonction de celle de leurs voisins
Le joueur peut choisir deux options : coopérer ou trahir	La cellule a deux états : vivante ou morte
stratégies	règles

TABLE 1 – Version traduite du tableau des différences entre le *DP* et le *JdlV*

6.4 Conclusions tirées de l'analyse

Des concepts intéressants ressortent de cette analyse, voici des concepts à retenir pour le développement du projet :

Système d'historique
Stratégies
Comparaisons entre stratégies
Grille "d'échiquier" pour cellules voisines
Imitation-of-the best

7 Analyse fonctionnelle

7.1 Maquette de l'interface

Dans cette partie du document, les diverses interfaces graphiques de l'application seront détaillées et expliquées.

7.1.1 Interactions entre fenêtres

La fenêtre principale de l'application (en bleu) possède deux modes de fonctionnements : Le mode standard et le mode étendu. C'est depuis cette fenêtre que l'on peut accéder aux divers menus et fenêtres de l'application.

Dans le cas du schéma ci-dessous, on considère la vue principale et la vue étendue comme deux vues différentes. Pour basculer de la vue principale à la vue étendue ou inversement, on actionne un *switch* se trouvant en bas à droite de la fenêtre.

Pour passer de la vue principale (ou étendue) à la fenêtre "à propos", on clique sur le bouton correspondant qui se trouve sur la barre de navigation.

Pour passer de la vue principale (ou étendue) à la fenêtre de paramétrage de la matrice des gains, on clique tout d'abord sur l'onglet "*Settings*" de la barre de navigation, puis sur l'option "*Payoff matrix*" du menu déroulant. Idem pour accéder aux paramètres de génération mais en cliquant sur l'option "*Generate new board*" du menu déroulant.

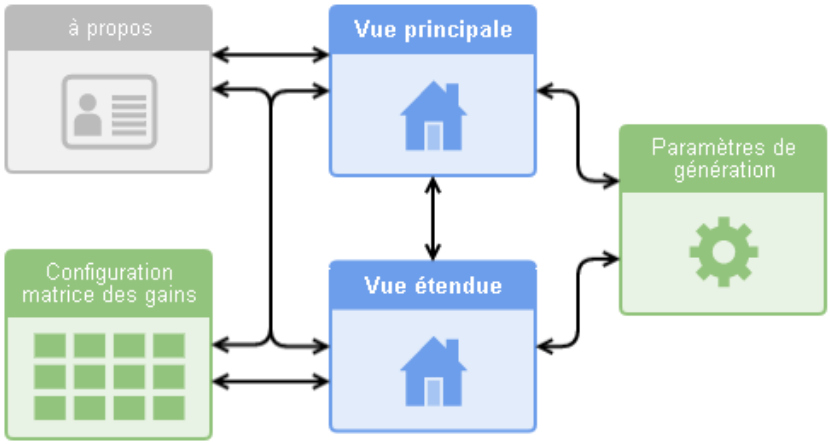


FIGURE 7 – Interactions entre fenêtres de l'application

7.1.2 Fenêtre principale

La fenêtre principale de l'application est composée de plusieurs parties :

Nom du composant	Utilité
La grille :	Composant affichant l'automate cellulaire.
Paramètres de taille :	Permet de modifier le nombre de ligne et colonnes de la grille.
Paramètres de vitesse :	Modifie la vitesse de *step* en mode d'exécution automatique
Bouton *step* :	Passe au temps t_{i+1} de l'automate cellulaire (avance d'un "pas").
Bouton *start* / *stop* :	Démarre ou arrête l'exécution automatique de la commande "*step*"
Bouton *clear* :	Efface le contenu de la grille.
Bouton *extended view* :	Bascule entre la vue principale et la vue étendue.

L'interface suivante est un croquis et il est possible que des fonctionnalités soient ajoutées à la version finale de l'application.

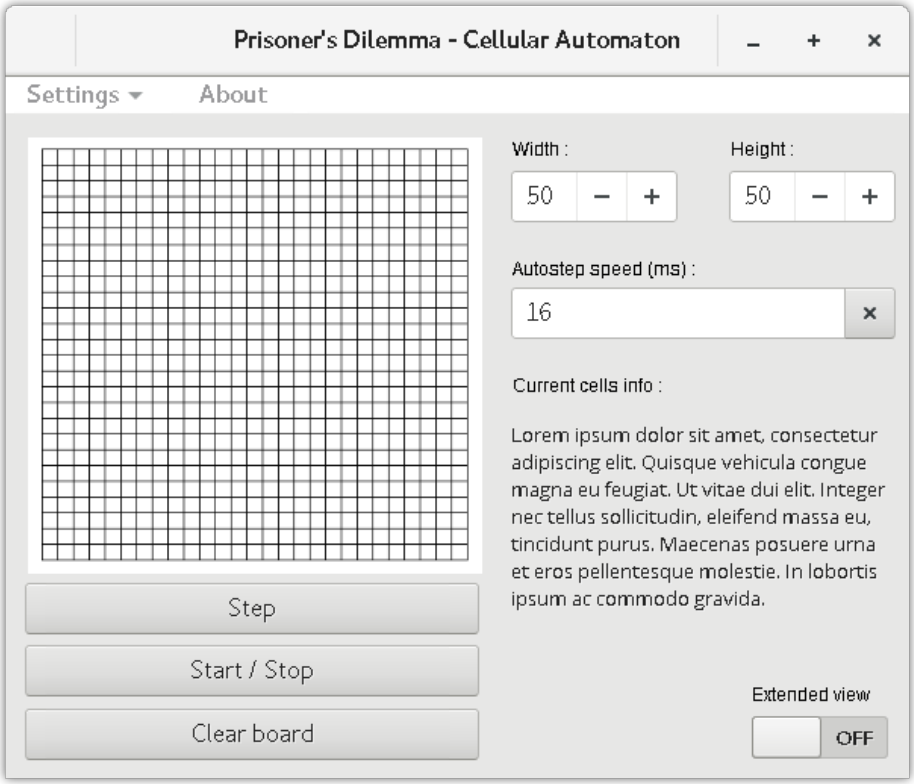


FIGURE 8 – Vue principale de l'application

7.1.3 Fenêtre principale (étendue)

La vue étendue est identique à la vue principale mais possède des graphiques supplémentaires permettant de visualiser plus facilement l'état actuel de l'automate cellulaire.

Voici des exemples de graphiques pouvant être implémentés dans l'application :

— Nombre de cellules "traîtres" par génération.
— Nombre de cellules "coopératives" par génération.
— Pourcentage de chaque stratégie utilisée.
— Stratégie et score maximum associé.
— etc...

Il est possible de basculer à tout moment de la vue étendue à la vue standard en déactivant le *switch* "*extended view*".

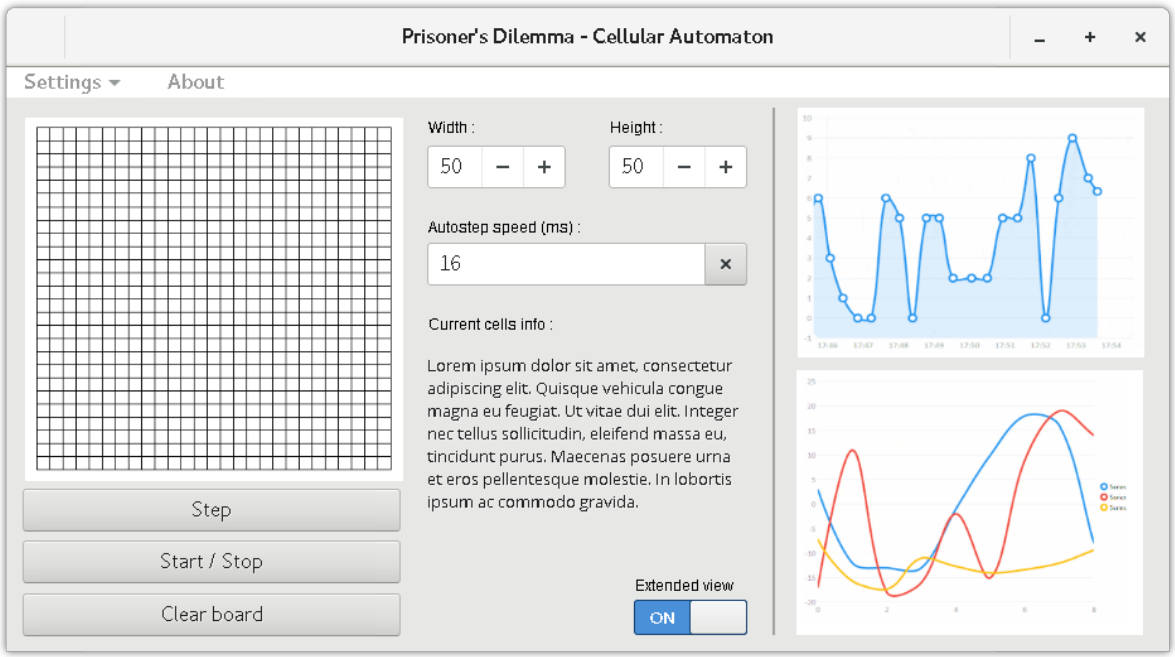


FIGURE 9 – Fenêtre étendue de l'application

7.1.4 Fenêtre principale, paramètres et "à propos"

Sur la fenêtre principale (ou étendue), une barre de navigation est présente en haut de page. Grâce à cette dernière, on peut accéder à un menu déroulant des paramètres de l'application (figure inférieure) et à la fenêtre à propos (figure supérieure).

Depuis le menu déroulant des paramètres, en cliquant sur le bouton "*Payoff matix*", on accède aux paramètres de la matrice de gains (voir "Matrice des gains"). En cliquant sur "*Generate new board*" : on accède aux paramètres de la génération d'un nouveau plateau (voir "Paramètres de génération").

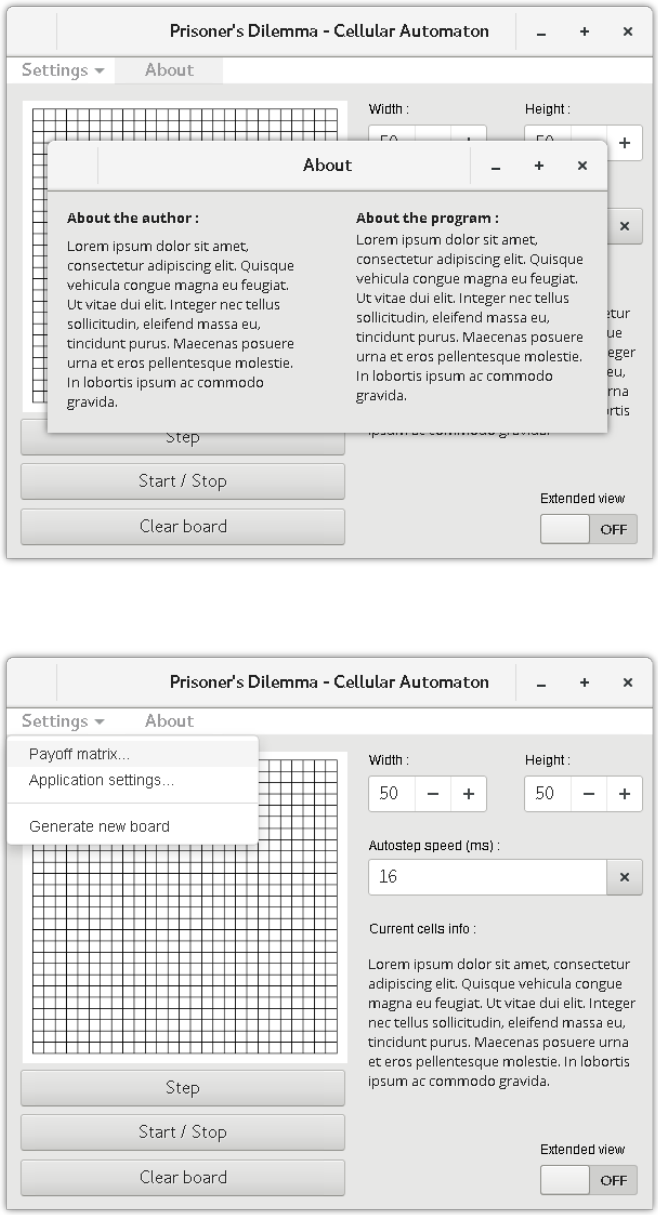


FIGURE 10 – Fenêtre "à propos" et accès aux paramètres de l'application

7.1.5 Matrice des gains

Sur la fenêtre des paramètres de la matrice des gains, on peut modifier différentes valeurs qui par la suite affecteront le comportement des cellules du plateau. Les paramètres présents sur la fenêtre correspondent aux quatre résultats pouvant être obtenus lors d'une partie du dilemme du prisonnier.

Les choix sont les suivants :

— *Reward payoff* (R)
— *Sucker's payoff* (S)
— *Temptation's payoff* (T) ou couramment appelé *Cheat's payoff* (C)
— *Punishment's payoff* (P)

On résume donc les valeurs de la matrice par les lettres R pour deux joueurs qui coopèrent, S pour le joueur s'étant fait trahir, T ou C pour le joueur ayant trahi et P pour les deux joueurs s'étant trahi.

On ne peut pas insérer n'importe quelles valeurs dans la matrice des gains. Les règles concernant les valeurs de la matrice sont les suivante :

$$T < R < P < S$$
$$2R < T + S$$

En cliquant sur le bouton "*OK*" se trouvant en bas de la fenêtre, on applique les modifications à la matrice des gains et on retourne sur la vue principale (ou étendue). Notez que le bouton "*OK*" de la page sera uniquement activé si les deux conditions citées précédemment sont respectées.

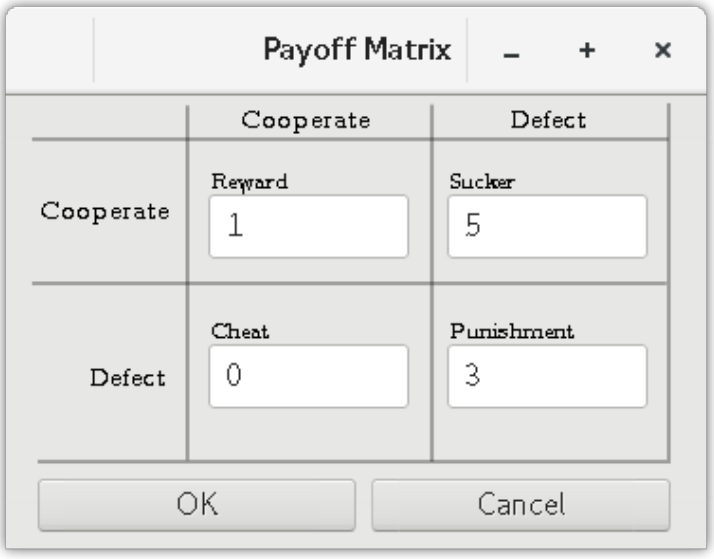


Figure 11 – Configuration de la matrice des gains de l'application

7.1.6 Paramètres de génération

La fenêtre "Paramètres de génération" donne la possibilité à l'utilisateur de générer un plateau de cellules avec un répartition aléatoire mais proportionnelle des stratégies.

On peut sélectionner les stratégies différentes à répartir sur le plateau à l'aide d'un champ texte. Voici un exemple correct de répartition des stratégies :

Random (RAND)	: 15%
Always Defect (AD)	: 15%
Always Cooperate (AC)	: 35%
Grim Trigger (GRIM)	: 35%
Total	: 100%

Le pourcentage total des stratégies sélectionnées doit impérativement être égal à 100%. Si ce n'est pas le cas, l'interface ne permettra pas à l'utilisateur de continuer (voir "Paramètres de génération, contrôles).

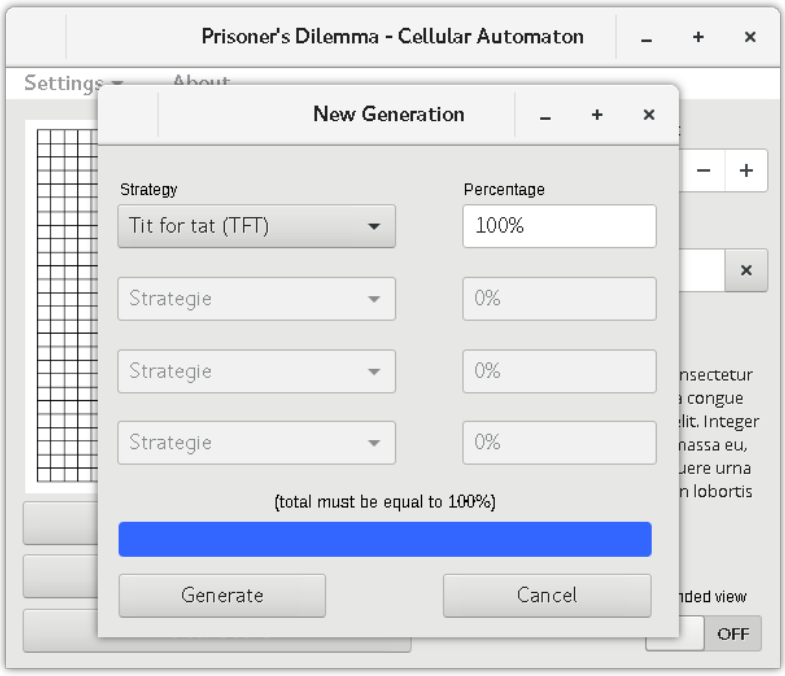


FIGURE 12 – Répartition aléatoire de cellules

7.1.7 Paramètres de génération, contrôles

Cette vue est ici pour démontrer les contrôles de la page "Paramètres de génération" empêchant les utilisateurs d'entrer des valeurs incorrectes. Notez que la barre de progression se trouvant en bas de la page est inférieure à 100%, empêchant ainsi l'accès à la génération du nouveau plateau.

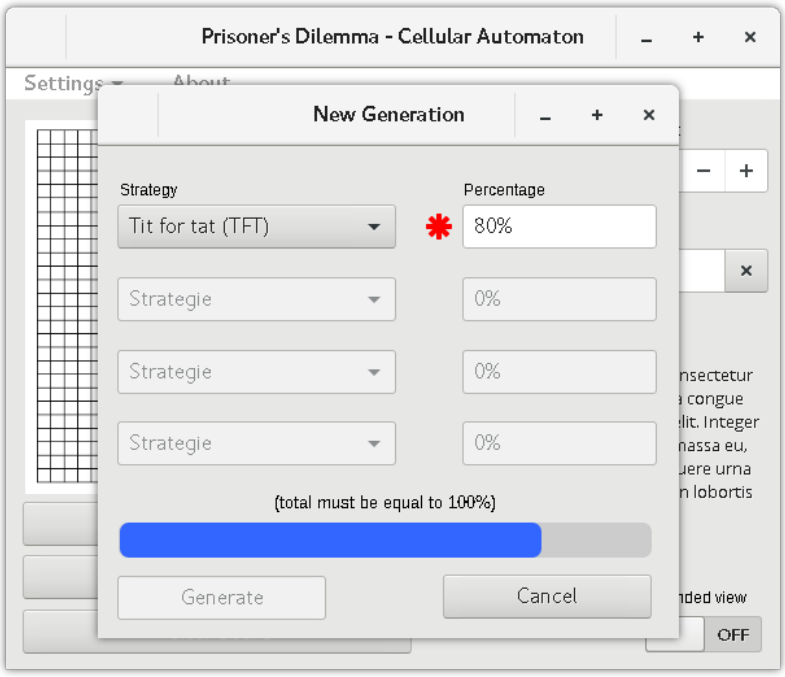


FIGURE 13 – Gestion des erreurs sur la génération aléatoire de cellules

7.2 Technologies utilisées

7.2.1 *LiveCharts*

LiveCharts est une bibliothèque C# permettant d'inclure des graphiques dynamiques dans des environnements *WPF* et *WinForms*. La bibliothèque permet l'utilisation de différents graphiques :

- *Cartesian charts* (Tableaux cartésiens)
- *Pie charts* (Camemberts)
- *Solid gauges* (Jauges)
- *Angular gauges* (Jauges angulaire)
- *Heatmaps & Geo maps* (Cartes)

Dans le cadre de ce projet, des tableaux cartésiens et des camemberts seront principalement utilisés pour représenter les données.

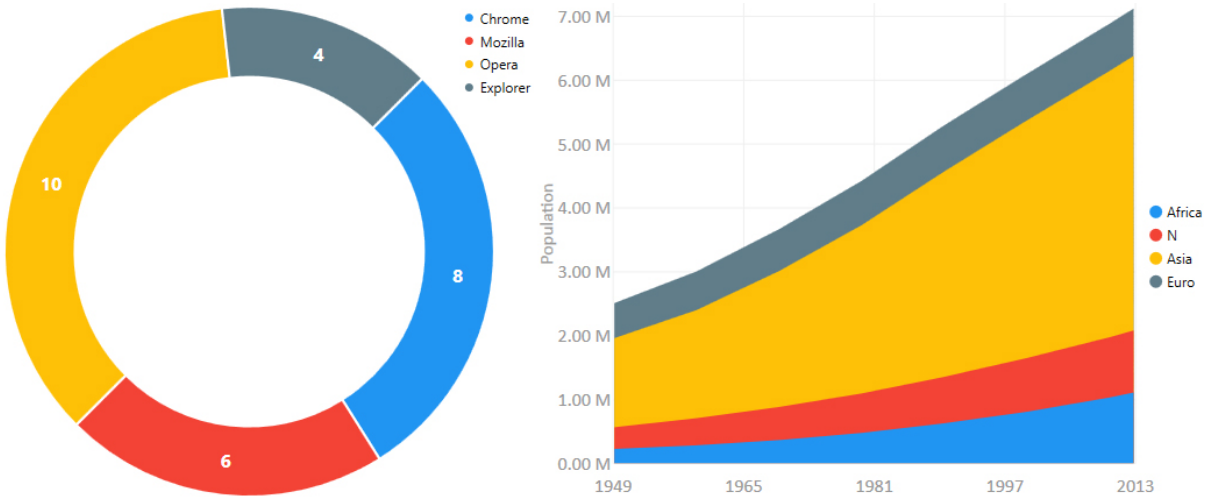


FIGURE 14 – Exemples de graphiques réalisés avec *LiveChars*

7.2.2 Tests unitaires

Les tests unitaires sont nécessaire pour vérifier le bon fonctionnement des différentes fonctions d'un projet. Traditionnellement, les tests sont réalisés avant le développement des fonctions. On appelle ce principe du "développement piloté par des tests" (ou *test driven developpement* en anglais).

Les tests unitaires facilitent différents aspects de la programmation :

- Faciliter le *debugging*
- Faciliter la maintenance
- Faciliter la rédaction de documentation

Les tests unitaires sont considérés comme de bonnes partiques lors du développement d'une application.

Dans le cadre de ce projet, toutes les fonctions du modèle seront testées à l'aide de tests unitaires.

7.2.3 *Design patterns*

Les *design patterns* (ou "patrons de conception") en français sont des solutions générales que l'on peut appliquer a un projet lors de sa conception. Un *design pattern* est reconnu comme une bonne pratique et est encouragé lors de la conception d'un logiciel.

Dans le cadre de l'automate cellulaire, le *design pattern* "*Strategy*" sera utilisé pour le mécanisme de stratégies des cellules. Ce design pattern permet de déléguer une méthode de notre classe à une classe spécialisée. Dans notre cas, la cellule délègue le choix de sa prochaine action à sa stratégie actuelle.

Voici un exemple UML du *design pattern* "*Strategy*" :

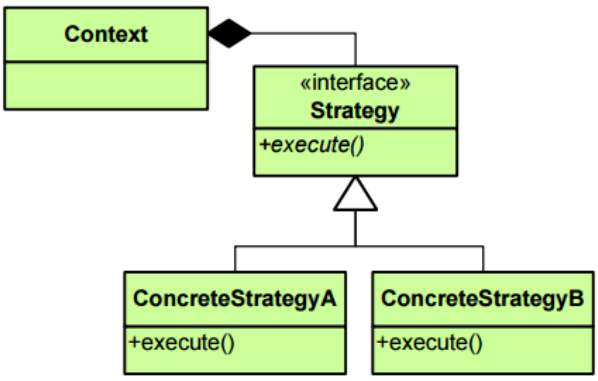


FIGURE 15 – Patron de conception "*Strategy*"

7.2.4 Sérialisation

La stérilisation permet d'enregistrer l'état de la mémoire d'une application dans un format spécifique. Dans le cadre de mon projet, la sérialisation sera implémentée au format "*.xml*". Le processus de récupération de ces données se nomme dé-sérialisation, et consiste a parcourir le fichier de données externe et de mettre en mémoire les différentes informations récupérées.

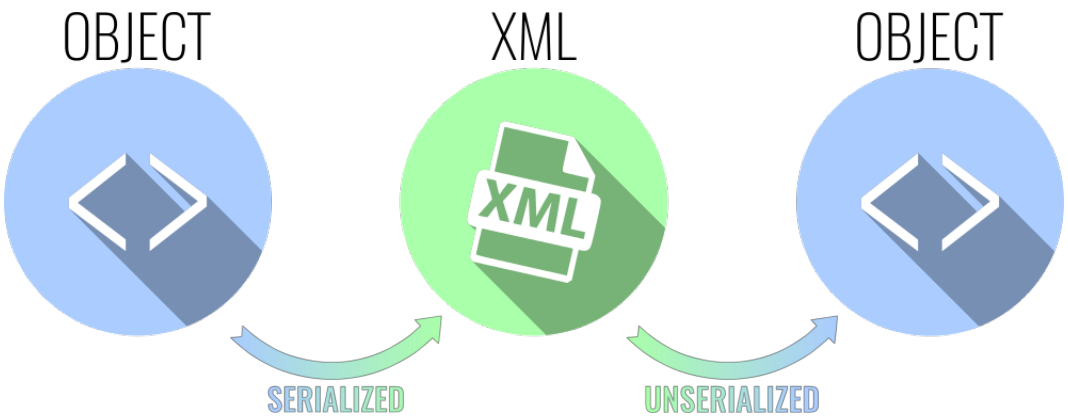


FIGURE 16 – Schéma de sérialisation "*.xml*"

7.3 Stratégies

Les joueurs du dilemme du prisonier utilisent de diverses stratégies[4][10] dans le but d'obtenir le meilleur score. Dans ce chapitre, les différentes stratégies adoptées par les cellules seront décrites en détail. On peut classer les stratégies en plusieurs groupes :

— Les stratégies originales (ex : *always cooperate*, *grim trigger*, etc...)

— *Tit-for-tat* et ses variantes. (ex : *tit-for-tat*, *reverse tit-for-tat*, etc...)

— Les stratégies de groupe (ex : *handshake*, *fortress*, etc...)

7.3.1 *Tit-for-tat*

Tit-for-tat est considéré comme la stratégie la plus simple ayant les meilleurs résultats[11]. Le nom *Tit-for-tat* implique une idée similaire à celle de "oeil pour oeil dent pour dent". Son fonctionnement est le suivant : on commence tout d'abord par coopérer, puis on observe le score de chacun de nos voisins. On copie par la suite la dernière action du voisin ayant obtenu le meilleur score.

Cette stratégie se retrouve couramment dans une situation de "*deadlock*", c'est-à-dire, une situation on l'on ne peut plus revenir à la coopération et ou trahir reste la seule bonne option.

7.3.2 *Tit-for-two-tats*

Tit-for-two-tats est une variante de *Tit-for-tat* ou on copie l'action du meilleur voisin uniquement si il fait deux fois la même action d'affilée. Si un voisin ne joue pas deux fois la même action d'affilée, on reste sur notre dernière action.

7.3.3 *Reverse tit-for-tat*

Reverse tit-for-tat est une variante de *Tit-for-tat*. Comme son nom l'indique, cette stratégie effectue les actions inverses de *Tit-for-tat*. Le joueur avec cette stratégie commence par trahir, puis il regarde la dernière action de son meilleur voisin et joue l'inverse au prochain tour.

7.3.4 *Always cooperate*

Comme son nom l'indique, un joueur avec cette stratégie *coopère* toujours.

7.3.5 *Always defect*

Comme son nom l'indique, un joueur avec cette stratégie *trahit* toujours.

7.3.6 *Random*

Comme son nom l'indique, un joueur avec cette stratégie joue de manière *aléatoire*. On pourrait comparer cette stratégie avec un joueur jouant a pile ou face à chaque tour pour déterminer sa prochaine action.

7.3.7 *Blinker*

Le mot *blinker* venant de l'anglais peut être traduit littéralement par "clignotant", illustrant le comportement de cette stratégie. Cette stratégie commence par coopérer, puis alterne entre trahison et coopération.

7.3.8 *Grim trigger*

La stratégie *grim trigger* est une stratégie que l'on peut dire "rancunière". Le joueur coopère tout le temps à la manière *always cooperate* jusqu'à qu'un joueur le trahisse. Après avoir été trahit, le joueur adopte une stratégie *always defect* et trahit de manière permanente.

7.3.9 *Handshake*

Handshake est la plus simple des stratégies de groupe. Elle consiste a identifier les voisins utilisant aussi la stratégie *Handshake*. Elle commence par "trahir, coopérer", si l'un de ses voisins effectue cette séquence, on coopère toujours, sinon on trahit toujours.

7.3.10 *Fortress*

Fortress est une stratégie de groupe visant a reconnaître les voisins utilisant aussi la stratégie *Fortress*. Elle est similaire à la stratégie *Handshake*. Elle commence par une séquence "trahir, trahir, coopérer", si l'un de ces voisins effectue cette même séquence, on la considère comme un "allié". Après avoir trouvé un "allié", on coopère jusqu'à la fin. Si on ne trouve pas "d'allié", on continue la séquence "trahir, trahir, coopérer".

7.3.11 *Southampton Group Strategy* (SGS)

Cette stratégie est similaire a *Fortress* et *Handshake* ; elle essaie d'identifier des voisins ayant la même stratégie avant d'effectuer une série d'action apportant un nombre maximal de points.

Dans le cas de *southampton group strategy*, elle commence par jouer une séquence de 5 à 10 mouvements prédéfinis au début de la partie. Après avoir reconnu d'autres cellules utilisant *southampton group strategy*, les cellules élisent un "maître" et le reste adoptent le comportement "d'esclave". La cellule "maître" trahit tout le temps et les cellules voisines "esclaves" coopèrent pour assurer le maximum de points au "maitre". Si une cellule *southampton group strategy* n'arrive pas a identifier des "aliés", elle trahit le reste de la partie.

7.3.12 *Pavlov*

Pavlov est une stratégie dite *heuristique* ou *rule-based* en anglais. Elle consiste a identifier la stratégie de ses voisins à l'aide de règles prédéfinies. Les stratégies des voisins sont classées dans quatre groupes :

- *Cooperative* (coopératif)
- *Always defects* (trahit toujours)
- *Tit-for-tat*
- *Random* (aléatoire)

Les six premiers tours de la partie sont consacrés à l'analyse des voisins, pendent cette période, *Pavlov* joue de manière identique à *Tit-for-tat*. Si l'adversaire ne commence pas a trahir dans ces tours, on l'identifie en tant que coopératif, *Pavlov* adopte donc une stratégie *Tit-for-tat*. Si l'adversaire trahit plus de quatre fois sur six, il est identifié en tant que *traitre* (trahit toujours) et *Pavlov* adopte une stratégie *always defect*. Si un voisins trahit exactement trois fois sur six, elle est identifiée en tant que *tit-for-tat* et *Pavlov* adopte donc une stratégie *tit-for-two-tats* pour essayer de coopérer avec *tit-for-tat*. Si un adversaire ne rentre pas dans ces catégories, on la classifie en tant que *random* (aléatoire) et *Pavlov* joue *always defect*. Les voisins peuvent cependant changer leurs actions, pour contrer ce mécanisme, *Pavlov* ré-évalue ses voisins chaque six tours.

8 Analyse organique

8.1 Diagramme de classe

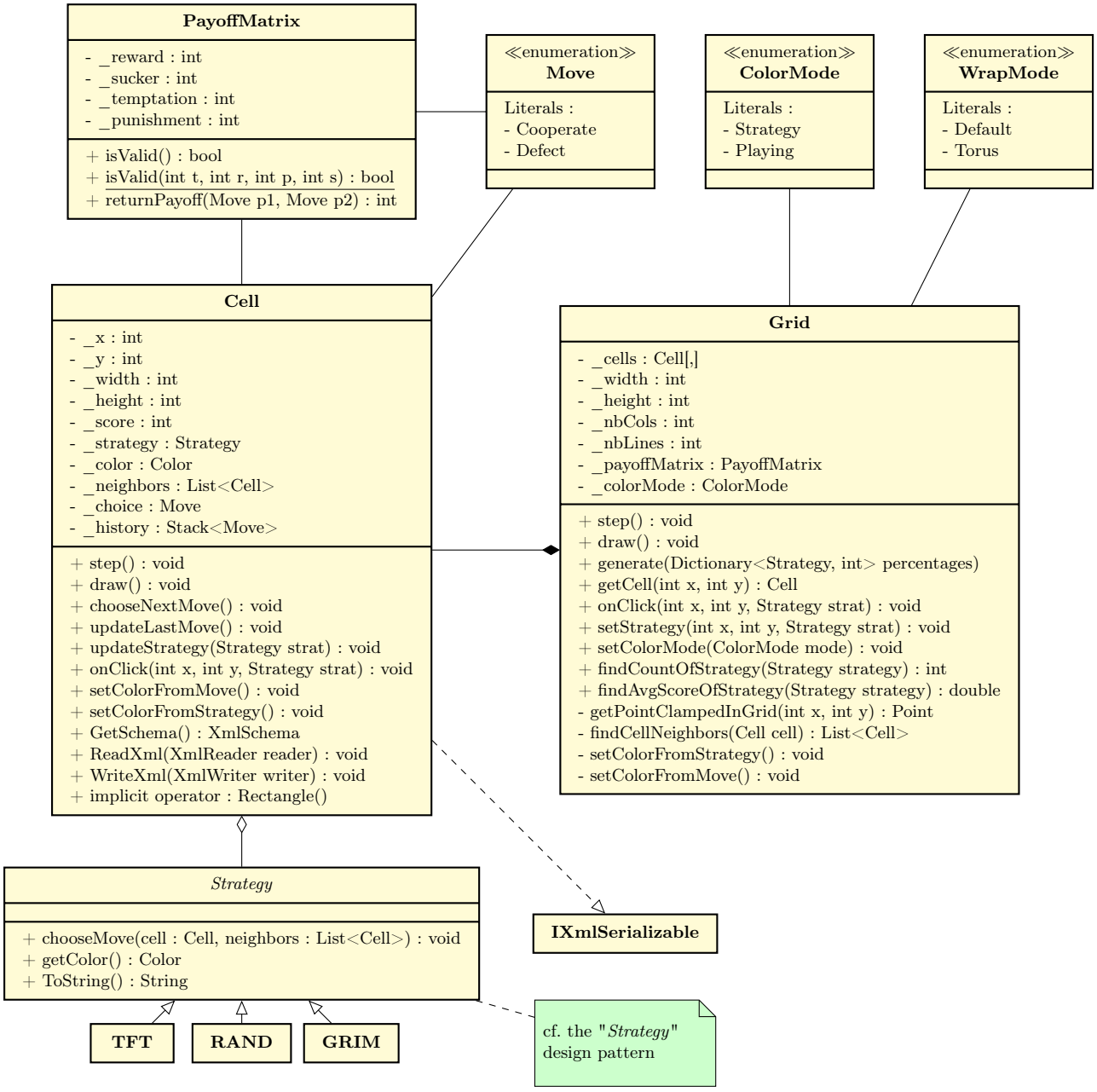


FIGURE 17 – Modèle UML de l'automate cellulaire du dilemme du prisonnier

8.2 Conventions de codage

Voici les conventions de codage respectées par toutes les classes de l'application.

8.2.1 En-têtes

Chaque classe de l'application possède une en-tête. Dans cette en tête, on résume la fonction de la classe, son auteur ainsi que sa date de création.

```
/*
    Class           :   Name.cs
    Description     :
    Author          :   SEEMULLER Julien
    Date            :   DD.MM.YYYY
*/
```

8.2.2 Commentaires

Chaque fonction de l'application est commentée. Le commentaire possède une courte explication de l'utilité de la fonction ainsi qu'une explication des paramètres si il y en a.

```
/// <summary>
/// Use of function
/// </summary>
/// <param name="param1">Use of the first parameter</param>
/// <param name="param2">Use of the second parameter</param>
/// <param name="param3">etc...</param>
```

8.2.3 Structure d'une classe

Chaque classe est découpée en six parties. En haut de classe se trouve l'en-tête, suivi par les *usings* et finalement suivi par le contenu de la classe. La classe est séparée en quatre parties à l'aide de régions (`#region`) : les champs, les propriétés, les constructeurs et les méthodes.

Voici un exemple de structure de code :

```
/* Header  */

using System;
using System.Collections.Generic;
[...]

public class myClass {
    #region fields
    #endregion

    #region properties
    #endregion

    #region constructors
    #endregion

    #region methods
    #endregion
}
```

8.3 Classes de l'automate cellulaire

Voici les différentes classes liés au fonctionnement de l'automate cellulaire :

- Classe `Cell`
- Classe `Grid`
- Classe `PayoffMatrix`
- Classe abstraite `Strategy`
- Énumérations `Move`, `ColorMode` et `WrapMode`

Dans ce chapitre vous trouverez le résumé détaillé du fonctionnement et de l'utilité de chacune de ces classes.

8.3.1 Classe `Cell`

La classe `Cell` est l'élément principal peuplant la grille de l'automate cellulaire. Voici les champs de la classe cellule ainsi qu'une courte description :

Champ	Description
`x`	: Numéro de la colonne ou se trouve la cellule.
`y`	: Numéro de la ligne ou se trouve la cellule.
`width`	: Largeur de la cellule en *pixels*.
`height`	: Hauteur de la cellule en *pixels*.
`score`	: Nombre de jours en prison de la cellule au tour actuel.
`strategy`	: Stratégie actuelle de la cellule. Determine les actions de la cellule.
`color`	: Couleur actuelle de la cellule.
`neighbors`	: Références vers les cellules voisines de la cellule.
`payoffMatrix`	: Référence vers la matrice des gains utilisée pour calculer le score de la cellule.
`choice`	: Prochaine action de la cellule (voir Énumération `Move`).
`history`	: Liste de toutes les actions de la cellule.

Suivant la philosophie *tell don't ask*, la logique ainsi que les données sont stockées à l'intérieur de la cellule. Chaque cellule est "responsable" du bon déroulement des méthodes appelées par les autres composants. La cellule a donc connaissance de tous ses voisins ainsi que la matrice de gains à l'aide de références vers ces derniers. Voici les méthodes de la classe cellule :

Méthode	Description
`step()`	: Joue une partie avec les voisins de la cellule en utilisant le choix actuel.
`chooseNextMove()`	: Choisit le prochain choix de la cellule (trahir ou coopérer) grâce à sa stratégie.
`updateLastMove()`	: Ajoute la dernière action effectuée à l'historique
`draw()`	: Dessine la cellule sur l'élément graphique passé en paramètre.
`setColorFromMove()`	: Change la couleur de la cellule selon ses actions (ex : trahir = rouge)
`setColorFromStrategy()`	: Change la couleur de la cellule selon sa stratégie (ex : AC = vert)
`updateStrategy()`	: Remplace la stratégie de la cellule avec celle passée en paramètre.
`Rectangle()`	: Précédé par `implicit operator` $\rightarrow$ conversion implicite de cellule en rectangle.
`onClick()`	: Change la stratégie avec celle passée en paramètre.
`GetSchema()`	: Inutilisé. Imposé par l'interface IXmlSerializable.
`ReadXml()`	: Lit le contenu d'une cellule depuis un fichier XML. Imposé par IXmlSerializable.
`WriteXml()`	: Écrit le contenu d'une cellule dans un fichier XML. Imposé par IXmlSerializable.

8.3.2 Analyse des méthodes de la classe `Cell`

Méthode "step()"
La méthode `step()` est la fonction principale de `Cell`. Elle permet de jouer une partie du dilemme du prisonnier avec ses voisins. On peut résumer une partie du dilemme du prisonnier par l'interaction entre deux actions (énumération `Move`) et la récompense qu'elles apportent. Pour obtenir ses gains, elle se réfère donc à la matrice des gains actuelle à l'aide de la méthode `returnPayoff()` (voir classe `PayoffMatrix`).

Après avoir récupéré le score de chacune des parties, on récupère le meilleur score (score minimum) de ces dernières comme score représentatif. D'autres alternatives comme une moyenne du score ou encore une médiane [12] ont été testées mais elles ne permettaient pas une bonne représentation du score et faussaient les calculs de certaines stratégies.

Voici a quoi ressemble la méthode `step()` de la classe `Cell` :

```
1  public void step()
2  {
3      // Go and play with each of our neighbors
4      List<int> scores = new List<int>();
5      foreach (Cell neighbor in this.Neighbors)
6      {
7          // Play a game and store the result
8          scores.Add(PayoffMatrix.returnPayoff(this.Move, neighbor.Move));
9      }
10
11     // We get the best score of the cell
12     this.Score = scores.Min();
13
14     // Update the color of the cell
15     this.setColorFromMove();
16 }
```

Méthode "chooseNextMove()"
La méthode `chooseNextMove()` détermine la prochaine action de la cellule à l'aide de sa stratégie. La stratégie détermine en fonction du voisinage et de l'état actuel de la cellule, quelle action effectuer.

Voici a quoi ressemble la méthode `chooseNextMove()` de la classe `Cell` :

```
1  public void chooseNextMove()
2  {
3      this.Choice = this.Strategy.chooseMove(this, this.Neighbors);
4  }
```

Méthode "Rectangle()"
Pour simplifier le fonctionnement des méthodes `onClick()` et `draw()`, il est possible de convertir *implicitement* une cellule vers un objet `System.Drawing.Rectangle`. La classe cellule possède des propriétés `nbLines` et `nbCols` indiquant sa position dans la grille ainsi que des informations sur ses dimensions. Grâce a ces informations, on convertit des informations relative à la grille (`Grid`) vers des informations relatives à l'écran.

Voici comment cette conversion est effectuée en C# :

```
1  public static implicit operator Rectangle(Cell cell)
2  {
3      return new Rectangle(cell.X * cell.Width, cell.Y * cell.Height, cell.Width, ↩
           cell.Height);
4  }
```

Méthode "onClick()"

La méthode `onClick()` prends en paramètre une paire de coordonnées [x, y] et vérifie si ce point est à l'intérieur la cellule actuelle. Si c'est le cas, on change la stratégie de cette dernière avec celle passée en paramètre à l'aide de la méthode `updateStrategy()`.

Pour simplifier la détection du point dans la cellule, on convertit implicitement notre cellule en rectangle, puis on utilise la méthode `Rectangle.Contains()` de ce dernier.

Voici le fonctionnement de la méthode `onClick()` présent dans la classe `Cell` :

```
1  public void onClick(int x, int y, Strategy strat)
2  {
3      Rectangle hitbox = this;
4
5      // If we are the cell that is hit, update our strategy and clear it's history
6      if (hitbox.Contains(x, y))
7      {
8          updateStrategy(strat);
9      }
10 }
```

Méthode "draw()"

La méthode `draw()` de la cellule permet de dessiner une cellule à l'aide de ses coordonnées [x,y] et ses dimensions. On utilise la conversion implicite vers un rectangle pour simplifier ce procédé. On définit aussi la couleur de la cellule grâce à la propriété crée à cet effet. La taille de la bordure des cellules peut aussi être ajustée à l'aide d'une constante.

Voici à quoi ressemble cette méthode :

```
1  public void draw(Graphics g)
2  {
3      // Color of the cell
4      SolidBrush cellColor = new SolidBrush(this.Color);
5
6      // Border parameters (color, width)
7      Pen borderColor = new Pen(Color.Black, DEFAULT_BORDER_WIDTH);
8
9      // Draw the cell
10     g.FillRectangle(cellColor, this); // Implicitly converted as a rectangle
11     g.DrawRectangle(borderColor, this);
12 }
```

Méthode "updateStrategy()"

La méthode `updateStrategy()` est utilisée principalement par la fonction `onClick()`. Cette dernière à pour but de mettre à jour la stratégie d'une cellule, tout en assurant le bon fonctionnement du jeu après ce changement. Pour cela, on fait jouer un tour du dilemme du prisonnier uniquement à la cellule changeant sa stratégie, ce qui permet de rester synchronisé avec les autres joueurs.

On s'assure également d'effacer l'historique de la cellule après avoir changé de stratégie ; on considère une cellule changeant de stratégie comme une toute nouvelle cellule remplaçant l'emplacement de la dernière.

Voici le code permettant de changer la stratégie d'une cellule :

```
1  public void updateStrategy(Strategy strat)
2  {
3      // Change the strategy
4      this.Strategy = strat;
5
6      // Updates the cell's move with the new strategy
7      this.History.Clear();
8
9      // We play a game with our neighbors to sync with the current game
10     this.chooseNextMove();
11     this.updateLastMove();
12     this.step();
13 }
```

Sérialisation
Dans cette partie du document, les méthodes permettant de sérialiser un objet `Cell` seront décrites. Les différentes fonctions implémentées permettent à `Cell` d'être conforme à l'interface `IXmlSerializable`.

Méthode `"GetSchema()"`
Cette méthode est inutilisée et doit toujours renvoyer `"null"`, voici la documentation officielle MSDN a ce sujet :

> *"Cette méthode est réservée et ne doit pas être utilisée. Au moment d'implémenter l'interface IXmlSerializable, vous devez retourner la valeur null (Nothing en Visual Basic) à partir de cette méthode. En revanche, si vous devez spécifier un schéma personnalisé, appliquez XmlSchemaProviderAttribute à la classe."*
>
> - MSDN

Voici donc le code réalisé pour cette méthode :

```
1 public XmlSchema GetSchema()
2 {
3     return null;
4 }
```

Méthode `"ReadXml()"`
La méthode `ReadXml()` permet de récupérer les informations d'une cellule depuis un fichier "*.xml*" sérialisé. Pour cela, on parcourt chaque propriété d'une cellule à l'aide de la fonction `reader.Read()`, jusqu'à arriver à la fin du document.

Voici le code permettant de récupérer des données depuis un fichier "*.xml*".

```
1  public void ReadXml(XmlReader reader)
2  {
3      reader.Read(); // Skip the beggining tab
4      if (reader.Name == "X")
5      {
6          reader.Read(); // Read past the name tag
7          this.X = int.Parse(reader.Value);
8          reader.Read(); // Read past the value
9
10     }
11     reader.Read(); // Read past the closing tag
12     // repeat this process for every value...
13 }
```

Méthode `"WriteXml()"`
A l'inverse de la méthode `"ReadXml()"`, la méthode `"WriteXml()"` permet d'écrire le contenu d'une cellule au format "*.xml*". Pour cela, on entoure chaque propriété de la cellule par une balise.

Voici a quoi ressemble le code de cette méthode :

```
1 public void WriteXml(XmlWriter writer)
2 {
3     // Write the content of the cell to xml format
4     writer.WriteStartElement("X");
5     writer.WriteString(this.X.ToString());
6     writer.WriteEndElement();
7
8     // repeat this process for every value...
9 }
```

8.3.3 Classe `Grid`

La classe `Grid` est le composant principal de l'automate cellulaire, elle est composée de cellules, possède une taille variable et un nombre d'éléments variables. Elle est aussi utilisée pour récupérer des données pour les différents graphiques de l'application à l'aide de méthodes comme "`findAvgScoreOfStrategy()`" ou encore "`findCountOfStrategy()`". Voici chacun des champs présents dans la classe `Grid` ainsi qu'une courte description :

Champ	Description
`cells`	: Tableau a deux dimensions (x, y) contenant les cellules.
`width`	: Largeur de la grille en *pixels*.
`height`	: Hauteur de la grille en *pixels*.
`nbCols`	: Nombre de colonnes de la grille.
`nbLines`	: Nombre de lignes de la grille.
`payoffMatrix`	: Matrice de gains du plateau. Distribué par la suite à toutes les cellules.
`colorMode`	: Définit si les couleurs affichées sur le plateau représentent les actions ou les stratégies des cellules.

Voici les méthodes de la classe `Grid` accompagnées d'une courte description :

Méthode	Description
`step()`	: Progresse vers l'état suivant de la grille. ($t_i = t_{i+1}$)
`draw()`	: Dessine les cellules et la grille sur l'élément graphique passé en paramètre.
`generate()`	: Génère une grille aléatoirement à l'aide de paires de stratégies et pourcentages.
`getCell()`	: Récupère une cellule grâce aux coordonnées $[x, y]$ passées en paramètre. Fonctionne de manière torique si une cellule est hors grille.
`getPointClampedInGrid()`	: Récupère un point grâce aux coordonnées $[x, y]$ passées en paramètre. Fonctionne de manière torique. Utilisé par `getCell()`.
`findCellNeighbors()`	: Renvoie les voisins d'une cellule passée en paramètre.
`onClick()`	: Active la méthode "`onClick()`" de chaque cellule du plateau.
`setStrategy()`	: Comme `onClick()`, mais avec les coordonnées de la grille ($\neq$ pixels).
`setColorMode()`	: Change le mode de couleur du plateau. (Voir Énumération `ColorMode`).
`setColorFromStrategy()`	: Méthode *privée*. Active la méthode "`setColorFromStrategy()`" de chaque cellule.
`setColorFromMove()`	: Méthode *privée*. Active la méthode "`setColorFromMove()`" de chaque cellule.
`findCountOfStrategy()`	: Renvoie le nombre de cellules du plateau ayant une stratégie identique à celle passée en paramètre.
`findAvgScoreOfStrategy()`	: Trouve le score moyen d'une stratégie passée en paramètre dans le plateau.
`saveData()`	: Sauvegarde la grille dans un format sérialisé.
`loadData()`	: Charge la grille depuis un fichier au format sérialisé.

8.3.4 Analyse des méthodes de la classe `Grid`

Méthode "`step()`"
La méthode `step()` s'occupe de faire progresser le plateau dans le temps. Une particularité de `step()` est que chaque action effectué par la fonction doit être effectuée impérativement l'une après l'autre. On utilise trois boucles séparées dans la fonction pour s'assurer que les états des historiques et des actions des cellules restent synchronisées. On évite ainsi que le choix actuel d'une cellule affecte celui d'une autre.

On commence par mettre à jour l'historique de la cellule (si une partie à été jouée précédemment). Puis, on choisit toutes les actions des cellules à l'aide de la fonction `Cell.chooseMove()`. Finalement, on lance une partie avec chaque cellule (`Cell.step()`).

Voici à quoi ressemble ce procédé :

```
1  public void step()
2  {
3      // Store each of the cell's last move
4      foreach (Cell cell in this.Cells)
5      {
6          cell.updateLastMove();
7      }
8
9      // Choose each of the cell's next move
10     foreach (Cell cell in this.Cells)
11     {
12         cell.chooseNextMove();
13     }
14
15     // Step forward (play the game)
16     foreach (Cell cell in this.Cells)
17     {
18         cell.step();
19     }
20 }
```

Méthode "`draw()`"
La méthode `draw()` s'occupe de dessiner la grille de cellule. Elle commence par dessiner chaque cellule à l'aide de la fonction `Cell.draw()` de ces dernières. Puis, elle dessine un contour aux endroits susceptibles aux erreurs de dessin due aux arrondissements.

Voici à quoi ressemble cette méthode :

```
1  public void draw(Graphics g)
2  {
3      // Draw each cell
4      foreach (Cell cell in this.Cells)
5      {
6          cell.draw(g);
7      }
8
9      // Avoid drawing errors due to rounding
10     Pen borderColor = new Pen(Color.Black, Cell.DEFAULT_BORDER_WIDTH * 2);
11     g.DrawLine(borderColor, 0, this.Height, this.Width, this.Height);
12     g.DrawLine(borderColor, this.Width, 0, this.Width, this.Height);
13 }
```

Méthode "`generate()`"
La méthode `generate()` s'occupe de générer un nouveau plateau de cellules à partir de stratégies et leur pourcentage de répartition. La méthode de base prends comme paramètre un dictionnaire ayant des stratégies comme clé et des pourcentages comme valeurs. Cependant, il existe une surcharge de la fonction permettant l'utilisation de deux listes séparées.

Pour la génération du plateau, on crée une liste représentant la proportion de chaque stratégie. Par exemple, si notre plateau possède 60% de *Tit-for-tat*, on ajoute 60 stratégies *Tit-for-tat* à la liste. Après ce procédé, il ne reste qu'a tirer des stratégies aléatoirement dans la liste, ce qui produit une bonne répartition proportionelle des stratégies tout en restant aléatoire.

Voici la méthode de génération de plateau :

```
public void generate(Dictionary<Strategy, int> strategyAndPercentages)
{
    // Create a new random number generator
    Random rng = new Random();

    // Create a list of a hundred elements representing the repartition of strategies
    List<Strategy> strategyPopulation = new List<Strategy>();

    // Go through each possible strategy and percentage
    foreach (var strat in strategyAndPercentages)
    {
        // Fill the list with the current strategy the same number of times as the ↩
            percentage
        for (int i = 0; i < strat.Value; i++)
        {
            strategyPopulation.Add(strat.Key);
        }
    }

    // Go through each cell in the grid
    foreach (Cell cell in this.Cells)
    {
        // Choose a random strategy in the list and apply it to the current cell
        int rnd = rng.Next(strategyPopulation.Count);
        cell.updateStrategy(strategyPopulation[rnd]);
    }
}
```

Méthode "`getPointClampedInGrid()`"
La méthode `getPointClampedInGrid()` existe pour faciliter la récupération de cellules sur le plateau. Elle prend en paramètre une paire de coordonnées `[x, y]` et renvoie la position de cette dernière de manière torique. La méthode accepte donc les valeurs se trouvant en dehors du plateau (ex : -1) et renvoie la position correspondante (voir Énumérations → `WrapMode`).

Voici le code de la méthode `getPointClampedInGrid()` :

```
public Point getPointClampedInGrid(int x, int y)
{
    int newX = x;
    int newY = y;

    // Check if we are out of bounds width-wise
    if (newX >= this.NbCols)
    {
        newX = newX - Convert.ToInt32(this.NbCols);
    }
    if (newX < 0)
    {
        newX = Convert.ToInt32(this.NbCols) + newX;
    }

    // Check if we are out of bounds height-wise
    if (newY >= this.NbLines)
    {
        newY = newY - Convert.ToInt32(this.NbLines);
    }
    if (newY < 0)
    {
        newY = Convert.ToInt32(this.NbLines) + newY;
    }
    return new Point(newX, newY);
}
```

Méthode "getCell()"
La méthode `getCell()` utilise la fonction `getPointClampedInGrid()` pour renvoyer une cellule du plateau. Cela permet de récupérer des cellules en ignorant le fait que les coordonnées passées en paramètre débordent en dehors de la grille.

Voici le code permettant de récupérer une cellule du plateau de manière torique :

```
1  public Cell getCell(int x, int y)
2  {
3      // Find the corrisponding point in a toroidal fashion if we go out of bounds
4      Point point = getPointClampedInGrid(x, y);
5      int newX = point.X;
6      int newY = point.Y;
7
8      // Return the correct cell
9      return this.Cells[newY, newX];
10 }
```

Méthode "findCellNeighbors()"
La méthode `findCellNeighbors()` est utilisée en interne pour trouver les voisins les plus proches d'une cellule du plateau. Selon le mode d'interaction des cellules (voir énumération `WrapMode`), les voisins seront sélectionnés de manière différente.

En mode normal, uniquement les cellules se trouvant à coté de la cellule actuelle sont considéré comme voisins. En mode torique, on regarde tout autour de la cellule actuelle et on contourne le plateau de manière torique pour trouver les différents voisins.

La portée à laquelle on considère des cellules comme voisins peut être ajustées à l'aide d'une constante nommée "NEAREST_NEIGHBOR_RANGE".

Voici à quoi ressemble le code permettant de récupérer les voisins les plus proches d'une cellule.

```
1  public List<Cell> findCellNeighbors(Cell cell)
2  {
3      List<Cell> neighbors = new List<Cell>();
4
5      // Go all around the cell to find its neighbors
6      for (int y = cell.Y - NEAREST_NEIGHBOR_RANGE; y <= cell.Y + ↩
           NEAREST_NEIGHBOR_RANGE; y++)
7      {
8          for (int x = cell.X - NEAREST_NEIGHBOR_RANGE; x <= cell.X + ↩
               NEAREST_NEIGHBOR_RANGE; x++)
9          {
10             // Avoid our own cell
11             if (!((x == cell.X) && (y == cell.Y)))
12             {
13                 // Add the neighbor depending on the mode
14                 switch (this.WrapMode)
15                 {
16                     case WrapMode.Default:
17                         // In default mode, check if we are inside the grid
18                         if ((x >= 0) && (y >= 0) && (x < this.NbCols) && (y < ↩
                               this.NbLines))
19                         {
20                             neighbors.Add(this.getCell(x, y));
21                         }
22                         break;
23
24                     case WrapMode.Torus:
25                         neighbors.Add(this.getCell(x, y));
26                         break;
27                 }
28
29             }
30         }
31     }
32
33     return neighbors;
34 }
```

Méthode "`onClick()`"
La méthode `onClick()` appelle simplement la fonction `onClick()` de chaque cellule. Voir `onClick()` de la classe `Cell` pour plus d'informations. On peut également utiliser la méthode `setStrategy()` pour le même résultat mais en utilisant des coordonnées locales de la grille ($\neq$ pixels).

Voici le code permettant d'appeler la méthode `onClick()` de chaque cellule.

```
public void onClick(int x, int y, Strategy strat)
{
    foreach (Cell cell in this.Cells)
    {
        cell.onClick(x, y, strat);
    }
}
```

Méthode "`setColorMode()`"
La méthode `setColorMode()` prends une valeur d'énumération `ColorMode` en paramètre et change le mode de couleur des cellules en fonction de ce dernier. Le mode de couleur permet de différentier les stratégies en mode `Strategy` et permet de différentier les cellules traitres des cellules coopératrices en mode `Playing`.

Voici l'extrait du code permettant de changer de mode de couleur.

```
public void setColorMode(ColorMode mode)
{
    // Switch according to the mode
    switch (mode)
    {
        case ColorMode.Strategy:
            this.setColorFromStrategy();
            break;
        case ColorMode.Playing:
            this.setColorFromMove();
            break;
    }

    this.ColorMode = mode;
}
```

Méthode "`findCountOfStrategy()`"
La méthode `findCountOfStrategy()` est utilisée pour peupler un graphique de l'application avec des données. Cette fonction renvoie le nombre de fois qu'une stratégie passée en paramètre apparaît sur le plateau.

On parcourt toutes les cellules du plateau en comparant le type de la stratégie de la cellule actuelle au type de celle passée en paramètre. A chaque fois que les deux types sont identiques, on incrémente un compteur.

Voici le code de cette méthode :

```
public int findCountOfStrategy(Strategy strategy)
{
    int count = 0;

    foreach (Cell cell in this.Cells)
    {
        // Find every cell that has the same type as the current strategy
        if (strategy.GetType() == cell.Strategy.GetType())
        {
            count++;
        }
    }

    // Return the result rounded down to two decimal places
    return count;
}
```

Méthode "`findAvgScoreOfStrategy()`"
La méthode `findAvgScoreOfStrategy()` est similaire à la fonction `findCountOfStrategy()`. Elle est utilisée pour peupler l'un des graphique de l'application avec des données. Cette méthode renvoie le score moyen d'une du plateau au moment de l'appel de la fonction.

Elle parcourt toutes les cellules du plateau et compare le type de la stratégie passée en paramètre au type de la stratégie actuelle. Quand les deux types sont identiques, on ajoute le score de la cellule à une variable (somme des scores) et on incrémente un compteur (nombre de cellules).

Pour obtenir la moyenne, on fait le calcul suivant : $\frac{sommeScores}{nbCellules}$

Voici à quoi ressemble cette méthode :

```
public double findAvgScoreOfStrategy(Strategy strategy)
{
    double count = 0;
    int i = 0;

    foreach (Cell cell in this.Cells)
    {
        // Find every cell that has the same type as the current strategy
        if (strategy.GetType() == cell.Strategy.GetType())
        {
            // Increment the total score and the count
            count += cell.Score;
            i++;
        }
    }

    // Find the percentage from the count
    count = (count / i);

    // Return the result rounded down to two decimal places
    return Math.Round(count, 2);
}
```

Méthode "`saveData()`"
La méthode `saveData()` est utilisée pour sérialiser la grille au format "*.xml*". Il est actuellement impossible de sérialiser un tableau multidimensionnel comme le composant `Cells` de `Grid`. Pour contourner ce problème, on convertit le tableau multidimensionnel vers une liste de cellules (`List<Cell>`). Pour plus d'informations sur ces conversions, voir la partie dédiée à `ArrayExtensions`.

Après avoir converti notre grille en liste, on crée un un nouveau `XmlSerializer` en lui renseignant le chemin vers le fichier à créer. On utilise la fonction `Serialize()` de l'objet ce qui permet d'écrire le fichier "*.xml*". Finalement, on s'assure de fermer les *streams* (flux de données) ouverts lors de l'écriture du fichier.

Voici le code de cette fonction :

```
public void saveData(string path)
{
    this.SerializableCells = this.Cells.asList();
    FileStream fs = new FileStream(path, FileMode.Create);
    XmlSerializer xs = new XmlSerializer(typeof(Grid));
    xs.Serialize(fs, this);
    fs.Close();
}
```

Le chemin pointant vers l'endroit ou écrire est habituellement passée en paramètre de la fonction. Cependant une surcharge de la méthode permet de le sauvegarder dans un répertoire par défaut :

```
public void saveData()
{
    this.saveData(DEFAULT_DATA_FILEPATH);
}
```

Méthode "`loadData()`"
La méthode `loadData()` permet de charger le contenu d'une grille depuis un fichier sérialisé au format "*.xml*". Elle commence par récupérer toutes les informations du fichier à l'aide de la fonction `Deserialize()` de l'objet `XmlSerializer`, puis elle les stocke dans une nouvelle instance d'un objet `Grid`.

Après avoir récupéré les données, on reconvertit la liste de cellule au format d'un tableau multidimensionnel (voir classe d'extension `ArrayExtensions`). Après cela, on reconstruit la liste des voisins de chaque cellule. Finalement, on transfert toutes les données de la grille temporaire vers la grille actuelle.

Voici le code permettant de charger une grille depuis un fichier "*.xml*" :

```
1  public void loadData(string path)
2  {
3      Grid newGrid;
4
5      XmlSerializer xs = new XmlSerializer(typeof(Grid));
6      using (StreamReader rd = new StreamReader(path))
7      {
8          newGrid = xs.Deserialize(rd) as Grid;
9      }
10
11     // rebuild the neighbors
12     newGrid.Cells = newGrid.SerializableCells.asArrayOfArray(newGrid.NbLines, ↩
           newGrid.NbCols);
13     foreach (var cell in newGrid.Cells)
14     {
15         cell.Neighbors = newGrid.findCellNeighbors(cell);
16     }
17
18     // Set each of the values from the serialized data
19     this.Width = newGrid.Width;
20     this.Height = newGrid.Height;
21     this.NbCols = newGrid.NbCols;
22     this.NbLines = newGrid.NbLines;
23     this.Cells = newGrid.Cells;
24     this.PayoffMatrix = newGrid.PayoffMatrix;
25     this.WrapMode = newGrid.WrapMode;
26 }
```

Similaire à la méthode `saveData()`, une surcharge de la méthode permet de charger le fichier depuis le chemin par défaut.

```
1  public void loadData()
2  {
3      this.loadData(DEFAULT_DATA_FILEPATH);
4  }
```

8.3.5 Classe `PayoffMatrix`

La matrice des gains est le composant permettant de calculer les gains de chaque cellule après une interaction. Elle est composée de quatre champs correspondants à la matrice des gains du dilemme du prisonnier :

Champ	Description
`reward`	: La récompense pour deux cellules qui coopèrent. (*reward payoff* en anglais).
`sucker`	: La peine pour une cellule s'étant fait trahir. (*sucker's payoff* en anglais).
`temptation`	: La récompense pour trahir une cellule, souvent égal à "0". (*temptation payoff* en anglais).
`punishment`	: La punition pour deux cellules traitres. (*punishment payoff* en anglais).

Dans cette version du dilemme du prisonnier, le score d'une cellule correspond au nombre de jours passés en prison d'une cellule. En suivant cette logique, quelques règles doivent être changées pour assurer le bon fonctionnement de l'automate cellulaire.

Les conditions de validité d'une matrice de gains du dilemme du prisonnier sont traditionnellement $T > R > P > S$ et $2R > T + S$ (ou $T = temptation$, $R = reward$ etc...). La relation $R > P$ implique que la coopération mutuelle est supérieure à la trahison mutuelle et les relations entre $T > R$ et $P > S$ impliquent que la trahison est la stratégie "dominante". La condition $2R > T + S$ est utilisée uniquement lors d'un dilemme du prisonnier répété et indique que deux coopérations mutuelles d'affilée est plus rentable que l'alternat T ou S.

Ces conditions sont uniquement vraies si on compte les jours de prisons "évités" comme un score positif. Dans notre cas, le score représente le nombre de jours en prison ; un score élevé est donc signe d'une mauvaise performance au jeu.

En gardant cette logique en tête et en l'appliquant à notre cas, on obtient les règles de validités suivantes :

$$T < R < P < S$$
$$2R < T + S$$

Voici les fonctionnalités de la classe `PayoffMatrix` :

Méthode	Description
`returnPayoff()`	: Retourne le gain du joueur1 à partir de deux objets `Move`. (Voir "Énumération `Move`").
`isValid()`	: Renvoie `true` si la matrice de gains est valide et `false` dans le cas contraire. (Voir règles ci-dessus).

8.3.6 Analyse des méthodes de la classe `PayoffMatrix`

Méthode "`returnPayoff()`"
La méthode `returnPayoff()` permet de renvoyer le résultat de l'interaction entre deux joueurs dans une partie du dilemme du prisonnier. Elle prends en paramètre deux `Move` (voir énumération `Move`), un pour le joueur 1 et un autre pour le joueur 2. Avec les choix des deux joueurs, elle renvoie le score au joueur 1 après avoir effectué des comparaisons.

Voici le code de cette méthode :

```
1  public int returnPayoff(Move playerOneChoice, Move playerTwoChoice)
2  {
3      int payoff = 0;
4      switch (playerOneChoice)
5      {
6          case Move.Cooperate:
7              // Player 1 cooperates, Player 2 cooperates = Reward payoff
8              if (playerTwoChoice == Move.Cooperate)
9              {
10                 payoff = this.Reward;
11             }
12             // Player 1 cooperates, Player 2 defects = Sucker's payoff
13             if (playerTwoChoice == Move.Defect)
14             {
15                 payoff = this.Sucker;
16             }
17             break;
18         case Move.Defect:
19             // Player 1 defects, Player 2 cooperates = Temptation payoff
20             if (playerTwoChoice == Move.Cooperate)
21             {
22                 payoff = this.Temptation;
23             }
24
25             // Player 2 defects, Player 2 defects = Punishment payoff
26             if (playerTwoChoice == Move.Defect)
27             {
28                 payoff = this.Punishment;
29             }
30             break;
31     }
32     return payoff;
33 }
```

Méthode "`isValid()`"
La méthode `isValid()` permet de verifier la validité d'une matrice de gains. Il y a deux moyens d'accéder à cette méthode : depuis un objet `PayoffMatrix` (ex : `myMatrix.isValid()`), ou par l'appel de la fonction statique (ex : `PayoffMatrix.isValid(4,3,2,1)`). Dans les deux cas, la fonction renvoie la valeur `true` si la matrice respecte les règles du dilemme du prisonnier et `false` dans le cas contraire.

Voici le code permettant de vérifier la validité une matrice de gains *IPD* :

```
1  public static bool isValid(int t, int r, int p, int s)
2  {
3      bool result = false;
4
5      // First condition of validity
6      if ((t < r) && (r < p) && (p < s))
7      {
8          if (2 * r < t + s)
9          {
10             result = true;
11         }
12     }
13
14     return result;
15 }
16
17 // Overloaded function that allows isValid to be used on the current object
18 public bool isValid()
19 {
20     return PayoffMatrix.isValid(this.Temptation, this.Reward, this.Punishment, ↩
           this.Sucker);
21 }
```

8.3.7 Classe abstraite `Strategy`

La classe abstraite `Strategy` est un "moule" pour toutes les stratégies de l'application. Pour qu'une stratégie soit compatible avec une cellule, elle doit impérativement hériter de la classe `Strategy`. La classe `Strategy` ne possède pas de champs. Elle impose cependant à ses classes "enfants" d'implémenter deux méthodes : `chooseMove()` et `getColor()`. Une méthode `ToString()` est aussi présente et utilisée pour renvoyer le nom de la stratégie, ce dernier est directement récupéré du nom de la classe et mis en forme pour que les classes "enfants" n'aient pas à modifier la fonction.

Voici les méthodes de la classe `Grid` accompagnées d'une courte description :

Méthode	Description
`chooseMove()`	: Choisit la prochaine action d'une cellule passée en paramètre.
`getColor()`	: Renvoie la couleur correspondante à la stratégie.
`ToString()`	: Renvoie le nom de la stratégie

8.3.8 Analyse des méthodes de la classe `Strategy`

Méthode "chooseMove()"

La méthode `chooseMove()` n'est pas implémentée dans la classe `Strategy` mais est imposée à toutes ses classes "enfants". Cette méthode représente la logique de la cellule et renvoie la prochaine action que la cellule doit effectuer.

Voici le prototype de la méthode `chooseMove()` :

```
1 public abstract Move chooseMove(Cell cell, List<Cell> neighbors);
```

Méthode "getColor()"

La méthode `getColor()`

```
1 public abstract Color getColor();
```

Méthode "ToString()"

La méthode `ToString()` renvoie le nom de la stratégie actuelle. Comparé aux deux autres fonctions, `ToString()` n'est pas imposée aux classes enfants et est réalisé directement dans la classe `Strategy`. La convention de nommage des classes de stratégies est la suivante :

"Strat" + `NomDeStratégie` + ".cs"
ex : `StratTitForTat.cs`

On récupère le nom de la classe à l'aide de `this.GetType().Name`, puis on retire le mot "Strat" si il est présent. Finalement, on ajoute un espace après chaque majuscules pour reformater le *CamelCase*.

Des expressions régulières (*regex*) sont utilisées pour ces traitements, ce qui rends la structure robuste et adaptable a des classes n'étant pas forcement conforme à la convention de nommage.

```
1  public override string ToString()
2  {
3      // Get the name of the current class
4      string strategyName = this.GetType().Name;
5
6      // Filter the name (remove "Strat" and use spaces insted of CamelCase)
7      strategyName = Regex.Replace(strategyName, "(Strat)", "");
8      strategyName = Regex.Replace(strategyName, "([a-z])([A-Z])", "$1 $2");
9
10     return strategyName;
11 }
```

8.3.9 Énumérations

Les énumérations ne sont pas des classes mais il est important de détailler leur rôle et fonction dans le programme. Trois énumérations sont actuellement présentes dans le programme :

- `Move`
- `ColorMode`
- `WrapMode`

`Move` représente les actions qu'un joueur peut faire dans le dilemme du prisonnier. `Move` a donc deux états possibles : `Cooperate` (coopérer) et `Defect` (trahir).

`ColorMode` représente le mode de couleur actuel de l'automate cellulaire. Il existe deux modes de couleurs dans l'automate cellulaire : un mode permettant de voir quelles stratégies sont sur le plateau actuellement, et un mode permettant de voir les actions des cellules du plateau. `ColorMode` a donc deux états possibles : `Playing` (en train de jouer) et `Strategies` (stratégies).

`WrapMode` définit le mode d'interaction entre cellules. Il y a actuellement deux modes, un mode par défaut (`Default`) et un mode torique (`Torus`). Quand `WrapMode` est en mode `Torus`, les cellules se trouvant en bords de grille sont voisines avec celles se trouvant aux bords opposés comme si la grille était un Tore [13]. En mode défaut, les voisins d'une cellule se trouvent directement a coté d'elle. La figure ci-dessous représente les différents modes d'inteactions entre cellules :

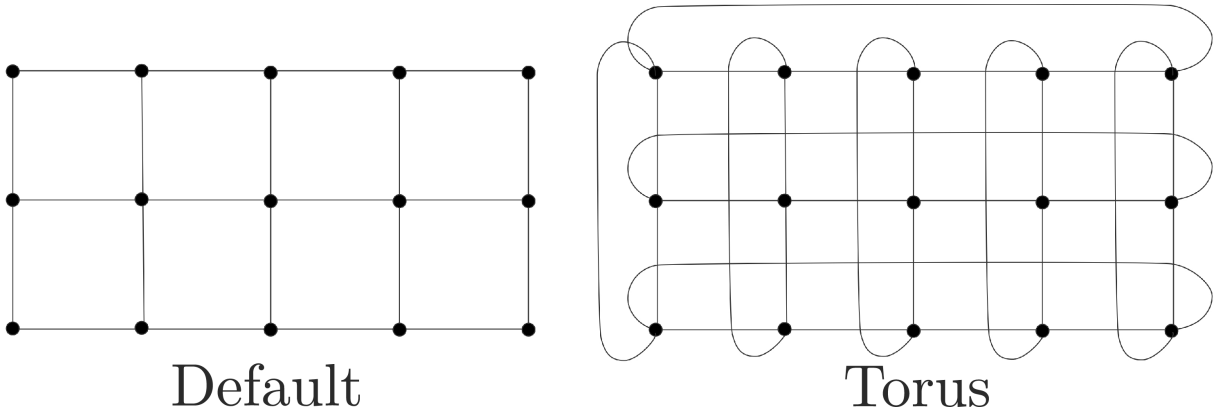


FIGURE 18 – Voisinage des cellules selon le "`WrapMode`"

8.4 Classes d'extensions

8.4.1 Classe `ColorExtensions`

La classe `ColorExtensions` à pour but de faciliter la conversion entre les objets `System.Media.Color` utilisé par *LiveCharts* et `System.Drawing.Color` utilisé par le reste de l'interface graphique et les cellules. Le seul moyen de passer d'un `System.Drawing.Color` à un `System.Media.Color` est de convertir la couleur au format hexadécimal. Pour ceci, plusieurs méthodes ont été réalisées :

- `ToHex(this Color c)`
- `ToHex(this Color c, byte transparency)`

La méthode `ToHex()` permet de convertir une couleur au format hexadécimal. Une surcharge de cette fonction permet de passer en paramètre la transparence désirée. Par exemple, pour récupérer le code hexadécimal de la couleur d'une des stratégies à environ 20% d'opacité, on peut appeler la fonction de la manière suivante :

```
1 string result = strategy.getColor().ToHex(51);
```

Il donc garder en tête que la transparence est codée sur un seul *byte* (0-255) lors de l'utilisation de la fonction.

8.4.2 Classe `ComboBoxExtensions`

La classe `ComboBoxExtensions` a été développée dans le but de rendre le choix de la stratégie par l'utilisateur plus agréable. La méthode `AddStrategies()` permet d'ajouter des stratégies à l'intérieur d'un composant `ComboBox` à partir d'une liste de stratégies. L'usage standard de cette fonction ressemblerait à :

```
1 List<Strategy> availableStrategies = new List<Strategy>
2 availableStrategies.Add(new StratTitForTat());
3 [...]
4 comboBox.AddStrategies(availableStrategies);
```

La méthode `DrawItem()` des `ComboBox` est remplacée par une version ajoutant un rectangle de couleur à coté de chaque élément de la liste. Ceci permet de voir à l'avance les couleurs des cellules placées sur la grille, et d'associer plus facilement une couleur à une stratégie dans l'esprit de l'utilisateur.

8.5 Classe `ArrayExtensions`

La classe `ArrayExtensions` existe pour faciliter la conversions en différents types de tableaux ; principalement, d'un tableau multidimensionnel à un liste et *vice versa*.

Voici les prototypes des fonctions de la classe `ArrayExtensions` :

```
1 // Converts a 2d array to a list.
2 public static List<object> asList(this object[,] inputArray)
3
4 // Converts a list to a 2d array.
5 public static object[,] asArrayOfArray(this List<object> inputArray, int nbLines, int ↩
      nbCols)
```

Voici un exemple d'utilisation de ces fonctions :

```
1 Cell[,] cellArray = new Cell[10, 10];
2 List<Cell> cellList = cellArray.asList();            // Converts to a list
3 [...]
4 Cell[,] cellArray2 = cellList.asArrayOfArray(10, 10) // Converts to a 2d array
```

8.6 Stratégies

Certaines stratégies connues du dilemme du prisonnier doivent être adaptées avant leur utilisation dans l'automate cellulaire. Les cellules jouent simultanément avec plusieurs voisins mais une seule action est choisie par tour. Cette contrainte nous force à ajuster certaines stratégies.

Dans ce chapitre, le fonctionnement interne de diverses stratégies sera décrit.

8.6.1 Stratégies simples

On classifie de stratégies simples les stratégies n'ayant pas besoin de tirer d'informations de l'état du jeu actuel. Des exemples de stratégies simples pourraient être *always cooperate*, *always defect* ou encore *blinker*.

Ces stratégies sont la plupart du temps implémentées en quelques lignes, en voici des exemples :

```
// Blinker
if (cell.History.Count \% 2 == 0)
     { result = Move.Cooperate; }
else { result = Move.Defect; }

return result;

// Always cooperate
return Move.Cooperate;

// Always defect
return Move.Defect;
```

8.6.2 *Tit-for-tat*

On peut résumer *tit-for-tat* à une stratégie imitant la dernière action de son adversaire. Dans le cas de l'automate cellulaire, une cellule possède plusieurs adversaires ce qui pose un problème : quel adversaire doit-on imiter ?

Dans cette implémentation de *tit-for-tat*, on choisit l'adversaire selon son score. L'adversaire ayant eu le meilleur résultat sera imité. Cette approche est tirée de l'un des travaux étudiés dans l'étude d'opportunité, et se base sur le concept de "*immitation of the best*".

Voici le code de la fonction `chooseMove()` de *tit-for-tat* :

```
public override Move chooseMove(Cell cell, List<Cell> neighbors)
{
    // Cooperates on first move, then copies his best openent
    Move result = Move.Cooperate;


    // If this wasn't our first round, we look at our neighbors
    if (cell.History.Count > 1)
    {
        // We initialise our variables with the first neighbor in the list
        result = neighbors[0].History.First();
        int min = neighbors[0].Score;

        foreach (Cell neighbor in neighbors)
        {
            if (min > neighbor.Score)
            {
                min = neighbor.Score;
                result = neighbor.History.First();
            }
        }
    }

    return result;
}
```

Notez que ce concept est appliqué à toutes les variantes de *tit-for-tat* de l'application.

8.6.3 *Grim trigger*

Grim trigger est une stratégie coopérant toujours avant d'avoir d'être trahi. Après être trahi, elle trahi tout le temps similaire à *always defect*. Pour illustrer ce principe de mémoire, un booléen est présent dans la classe garde en mémoire si la cellule à été trahie. Ceci est un bon exemple de la flexibilité de la structure, une classe peut avoir des champs et des fonctions a part et peut être considérée comme stratégie si elle implémente `chooseMove()` et `getColor()`.

Voici le code de `chooseMove()` dans cette stratégie :

```
1  public override Move chooseMove(Cell cell, List<Cell> neighbors)
2  {
3      // Starts by cooperating
4      Move result = Move.Cooperate;
5
6      // Check if we were betrayed in the past
7      if (WasBetrayed)
8      {
9          result = Move.Defect;
10     }
11     else
12     {
13         // If we didn't get betrayed yet, we look at our neighbors
14         if (cell.History.Count > 1)
15         {
16             // Look if we got betrayed by a neighbor after our first move
17             foreach (Cell neighbor in neighbors)
18             {
19                 if (neighbor.History.First() == Move.Defect)
20                 {
21                     // If we are betrayed, we switch to a "Always Defect" strategy
22                     this.WasBetrayed = true;
23                     result = Move.Defect;
24                     break;
25                 }
26             }
27         }
28     }
29
30     return result;
31 }
```

8.7 *Benchmark* des stratégies

Dans le but de trouver la meilleure stratégie, une partie de l'application permettant de tester les performances d'une stratégie à été réalisée. Depuis cette interface, l'utilisateur peut sélectionner une stratégie à tester, ainsi que le nombre de tours du dilemme du prisonnier à jouer.

L'inspiration pour cette fonctionnalité provient des tournois du dilemme du prisonnier organisés par Robert Axelrod (popularisés dans son livre : *The Evolution of Cooperation*[14]). Ces tournois ont étés utilisés précédemment dans le but d'analyser les différentes stratégies, leurs performances, ainsi que les critères nécessaires pour qu'une stratégie soit efficace.

Quand l'utilisateur lance la simulation, la stratégie est testée contre toutes les autres stratégies de l'application. La stratégie sélectionnée joue le nombre de parties défini par l'utilisateur contre chacune des stratégies.

Après la simulation terminée, l'utilisateur peut observer les résultats sur un graphique. Le score total de chaque parties contre chaque stratégies est représentée par un histogramme.

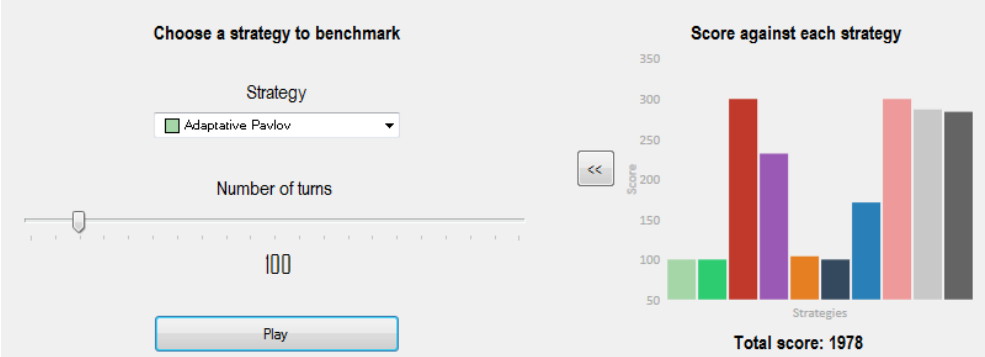


FIGURE 19 – Interface *benchmark*

Grâce à cette fonctionnalité, un graphique démontrant l'efficacité de chaque stratégies à été réalisé. Ce graphique fut réalisé en comparant chaque stratégie sur 100 tours du dilemme du prisonnier. Le score représente le nombre de jours passés en prison, le score le plus bas est donc le meilleur.

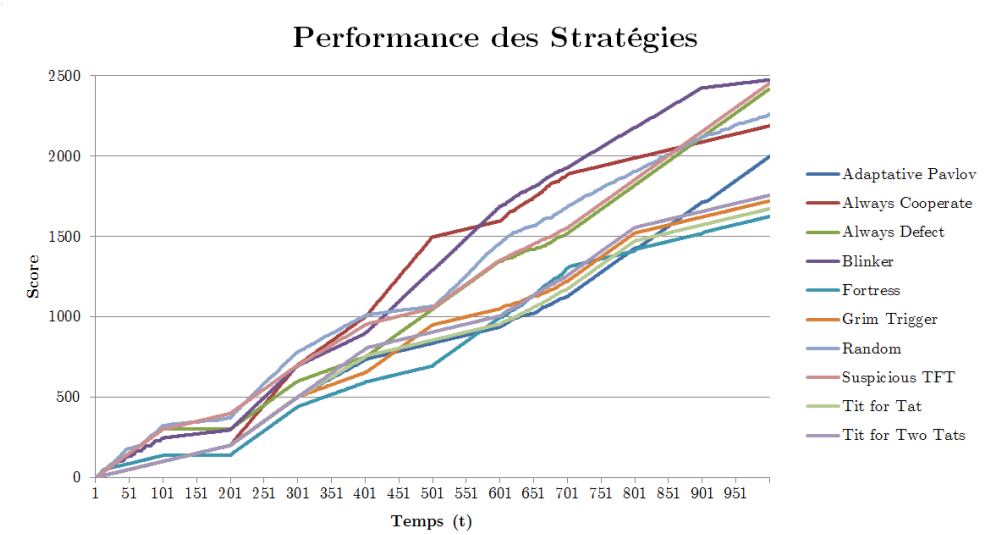


FIGURE 20 – Test des performances des stratégies

9 Tests

Tous les tests unitaires ont étés réalisés dans l'environnement de test de Visual Studio 2013. Cependant, il est actuellement impossible d'exporter les résultats de ces derniers depuis Visual Studio. Pour remédier à ce problème, j'ai développé un script *batch* permettant d'exporter les tests d'une solution au format "*.trx*" [1] à l'aide de la console de test Visual Studio (vstest.console.exe).

On lance simplement la console de test Visual Studio avec le lien vers notre fichier "*.dll*" de tests et "/Logger :trx" en paramètre. Les résultats seront automatiquement exportés vers un dossier nommé "TestsResults".

Voici le contenu du script *batch* permettant d'exporter les résultat des tests :

```
"C:\[...]\vstest.console.exe" "C:\[...]\PrisonersDilemmaCATests.dll" /Logger:trx
pause
```

Voici les résultats des tests unitaires des classes de l'automate du cellulaire :

Nom du test	Durée	Résultat
CellTests.ConvenianceConstructorTest	00 :00 :00.0002014	Réussite
CellTests.DesignatedConstructorTest	00 :00 :00.0083494	Réussite
CellTests.ImplicitConversionTest	00 :00 :00.0002080	Réussite
CellTests.onClickTest	00 :00 :00.0060866	Réussite
ColorExtensionsTests.ToHexTest	00 :00 :00.0004385	Réussite
ColorExtensionsTests.ToHexTransparentTest	00 :00 :00.0004142	Réussite
ColorExtensionsTests.ToRGBTest	00 :00 :00.0003641	Réussite
ComboBoxExtensionsTests.AddStrategiesTest	00 :00 :00.0247826	Réussite
GridTests.DesignatedConstructorTest	00 :00 :00.0006595	Réussite
GridTests.findCellNeighborsTest	00 :00 :00.0009470	Réussite
GridTests.getCellTest	00 :00 :00.0005589	Réussite
GridTests.getPointClampedInGridTest	00 :00 :00.0005565	Réussite
PayoffMatrixTests.DesignatedConstructorTest	00 :00 :00.0002266	Réussite
PayoffMatrixTests.isValidConvenianceTest	00 :00 :00.0003449	Réussite
PayoffMatrixTests.isValidStaticTest	00 :00 :00.0010773	Réussite
PayoffMatrixTests.returnPayoffTest	00 :00 :00.0002335	Réussite
StratAlwaysCooperateTests.chooseMoveTest	00 :00 :00.0004451	Réussite
StratAlwaysDefectTests.chooseMoveTest	00 :00 :00.0004544	Réussite
StratBlinkerTests.chooseMoveTest	00 :00 :00.0011629	Réussite
StrategyTests.CompareToTest	00 :00 :00.0004379	Réussite
StrategyTests.ToStringTest	00 :00 :00.0004382	Réussite
StratFortressTests.chooseMoveTest	00 :00 :00.0028304	Réussite
StratGrimTriggerTests.chooseMoveTest	00 :00 :00.0013625	Réussite
StratRandomTests.chooseMoveTest	00 :00 :00.0003097	Réussite
StratSuspiciousTitForTatTests.chooseMoveTest	00 :00 :00.0029031	Réussite
StratTitForTatTests.chooseMoveTest	00 :00 :00.0009945	Réussite
StratTitForTwoTatsTests.chooseMoveTest	00 :00 :00.0015303	Réussite

TABLE 2 – Résultat des tests unitaires

1. "*.trx*" représente un fichier de tests crée par Visual Studio

10 Estimation de l'apport personnel

Nom	Pourcent	Commentaire
Grille	100%	
Cellules	100%	
Matrice des gains	100%	
Stratégies	100%	Implémentation sur la base de courtes descriptions [4][10]
Classes d'extensions	70%	Modification d'algorithmes
Interface graphique	90%	Utilisation d'une ".dll" pour le composant "ToggleSwitch"
Graphiques	25%	Utilisation de la bibliothèque "LiveCharts"
Tests unitaires	100%	
Poster	100%	
Documentation	100%	

Table 3 – Apports personnel dans l'automate cellulaire du dilemme du prisonnier

11 Conclusions et perspectives

11.1 Projet

En conclusion, le cahier des charges à été entièrement rempli. De plus, quelques fonctionnalités ont pu être ajoutés. La fonction *benchmark* permet à l'utilisateur de tester une stratégie contre toutes les autres stratégies disponibles, dans l'optique de comparer ses résultats. Il est également possible de changer à tout moment le mode d'interaction des cellules. Deux modes sont disponibles, le mode normal et le mode torique (voir Énumération `WrapMode`).

11.2 Améliorations possibles

Le projet contient actuellement toutes les fonctions décrites dans le cahier des charges, mais n'est évidemment pas parfait. Plusieurs idées viennent à l'esprit lorsque l'on parle d'améliorations possibles :

- Amélioration des performances.
  - Graphique de la moyenne des scores.
- Changement de la logique du projet.
  - MVC
  - Huit choix pour huit voisins.

11.2.1 Graphique de la moyenne des scores

Le graphique affichant le score moyen de la cellule stocke actuellement tous les résultats de la partie actuelle, ce qui peut causer des ralentissement dans des parties longues. Pour corriger ce problème il est envisageable de créer un nouvel objet compatible avec le graphique de *LiveCharts*.

Pour cela, l'objet devrait spécifier la coordonnée "x" de chaque point au lieu d'être réparti automatiquement par le composant graphique. Cet objet posséderait donc le score et le numéro du tour ou le score à été prit. Après avoir implémenté cela, il ne resterait plus qu'à fixer un seuil pour le nombre d'éléments maximum a stocker et à retirer les éléments en trop.

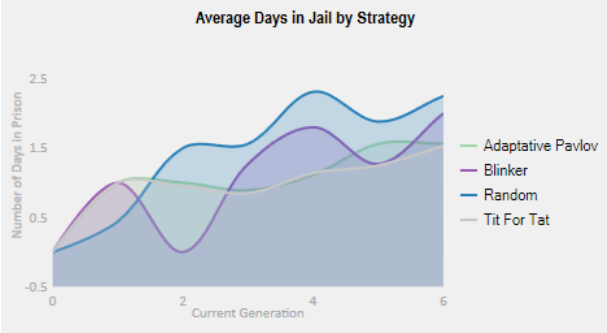


FIGURE 21 – Graphique causant des problèmes de performances

11.2.2 MVC

Actuellement le projet utilise une structure de type "`Model` → `View`" ce qui rends plus difficile la séparation du modèle de la vue. Utiliser une architecture MVC simplifierait l'implémentation du modèle dans un autre application.

11.2.3 Huit choix pour huit voisins

Actuellement, une seule action est choisi par une cellule, malgré le fait qu'une cellule possède huit voisins au maximum. Ce choix de conception permet de simplifier grandement la logique du jeu malgré les quelques imprécisions qu'elle peut causer au niveau des stratégies. En effet, certaines stratégies ont eu être adaptées pour fonctionner avec l'automate cellulaire. En adoptant une structure ou chaque cellule joue huit parties individuelles avec chacun de ces voisins, les stratégies peuvent rester plus fidèles à leur première implémentation, mais cela au coût de la simplicité. Ce concept pose des problèmes comme "comment définir la couleur de la cellule ?" mais il serait intéressant de l'implémenter pour observer les résultats obtenus.

11.3 Évolution de la coopération et fiabilité des stratégies

Dans "The Evolution of Cooperation" de Robert Axelrod[14], l'auteur définit différents critères pour qu'une stratégie puisse obtenir un bon score au dilemme du prisonnier. Les critères pour qu'une stratégie soit efficace sont les suivants :

Anglais		**Français**
Don't be envious.	→	Ne soyez pas envieux.
Don't be the first to defect.	→	Ne soyez pas le premier a trahir.
Reciprocate both cooperation and defection.	→	Réciproquez la coopération et la trahison.
Don't be too clever.	→	Ne soyez pas trop malin.

Table 4 – Réussite d'une stratégie

Le premier critère indique qu'il ne faut pas être envieux, cela implique que la stratégie ne dois pas chercher à avoir le meilleur score de manière égoïste pour réussir.

Le deuxième critère indique qu'il ne faut pas être le premier à trahir ses voisins. Cela assure une relation de "confiance" et empêche certaines stratégies de se "venger" au prochain tour.

Le troisième critère est qu'il faut réciproquer la trahison et la coopération. En résumé, il est important de coopérer mais il ne faut pas se laisser exploiter par des traîtres et il est nécessaire de trahir si l'adversaire trahit de manière répétée.

Le quatrième critère est qu'il ne faut pas être trop malin. L'auteur fait probablement référence à la stratégie *tit-for-tat* étant l'une des stratégies les plus simples et pourtant la gagnante de plusieurs tournois du dilemme du prisonnier. Ce critère indique aussi que des stratégies plus complexes ne seront pas forcement plus efficaces cela du à la nature du jeu. Une stratégie analysant son adversaire pour déterminer quelle action faire ne sera pas forcement plus efficace qu'une stratégie simple comme *tit-for-tat*.

Dans le cas de l'automate cellulaire du dilemme du prisonnier, ces critères s'appliquent toujours. Cependant, le troisième critère est beaucoup plus difficile à appliquer. Chaque cellule fait un seul choix pour jouer contre ses huit voisins. Cette condition implique que la coopération devient beaucoup plus difficile à rétablir. En effet, si une cellule décide de coopérer, il est nécessaire que la *totalité* de ses voisins coopère pour s'assurer que l'on se fasse pas exploiter par des cellules traîtres. C'est à cause de cette subtilité que des stratégies comme *tit-for-tat* ou *pavlov* ne décident de pas rétablir la coopération.

On peut donc tirer plusieurs conclusions de ces résultats. Du à la nature de l'automate cellulaire, il est difficile de rétablir la coopération après la trahison. De plus, les stratégies simples (*always cooperate*, *blinker*, etc...) ne sont pas une bonne approche et il est nécessaire d'utiliser les informations du plateau pour obtenir de meilleurs scores. Finalement, les stratégies visant à avoir un maximum de points (envieuses) sont plus performantes que dans le dilemme du prisonnier standard (un contre un), cela du à la disposition du jeu (un contre huit) et à la méthode d'attribution des scores (maximisé).

11.4 Perspectives

Grâce à ce projet, j'ai pu m'immerger dans un sujet qui m'était inconnu. Il m'aura permit d'appliquer différents concepts appris en classe, tels que : les *design patterns*, le *test driven developpement* ou encore la sérialisation. Il m'a également permit de renforcer mes compétences en documentation avec LaTeX.

Si je venais à travailler à nouveau sur ce projet, je pense changer la structure du programme pour que chaque cellule puisse choisir une action par voisin.

Finalement, ce projet m'a donné envie de me plonger dans le domaine de la théorie des jeux et de finir la lecture du livre de Robert Axelrod : "L'Évolution de la Coopération".

12 Sources

Références

[1] Wikipedia. *Jeu de la vie.*
URL : https://en.wikipedia.org/wiki/Conway's%5C_Game%5C_of%5C_Life.

[2] Wikipedia. *Dilemme du prisonnier.*
URL : https://en.wikipedia.org/wiki/Prisoner's%5C_dilemma.

[3] Wikipedia. *Matrice des gains.*
URL : https://fr.wikipedia.org/wiki/Matrice%5C_des%5C_gains.

[4] Wayne Davis. *Stratégies iterated prisoners dilemma.*
URL : http://www.iterated-prisoners-dilemma.net/prisoners-dilemma-strategies.shtml.

[5] Ramón Alonso-Sanz. *Dilemme du prisonnier, automate cellulaire.*
URL : http://rspa.royalsocietypublishing.org/content/470/2164/20130793.

[6] Livecharts. *Homepage.*
URL : https://lvcharts.net/.

[7] Alonso-Sanz Ramón. *A Quantum Prisoner's Dilemma Cellular Automaton.*
URL : http://rspa.royalsocietypublishing.org/content/royprsa/470/2164/20130793.full.pdf.

[8] Alves Pereira Marcelo. *Prisoner's Dilemma in One-Dimensional Cellular Automata.*
URL : https://arxiv.org/pdf/0708.3520.pdf.

[9] Zbieć Katarzyna. *The Prisoner's Dilemma and The Game of Life.*
URL : logika.uwb.edu.pl/studies/download.php?volid=19&artid=kz.

[10] Wayne Davis. *Strategies for IPD.*
URL : http://www.prisoners-dilemma.com/strategies.html.

[11] Weisstein Eric. *Tit-for-tat.*
URL : http://mathworld.wolfram.com/Tit-for-Tat.html.

[12] Wikipedia. *Médiane (statistiques).*
URL : https://fr.wikipedia.org/wiki/M%5C%C3%5C%A9diane_(statistiques).

[13] Wikipedia. *Tore (forme).*
URL : https://fr.wikipedia.org/wiki/Tore.

[14] Robert Axelrod. *The Evolution of Cooperation.* Basic Books, 1984. ISBN : 0465021220.

Table des figures

Liste des tableaux

CFPT-INFORMATIQUE

Travaux de Diplômes 2017

Code Source

Automate Cellulaire

Julien SEEMULLER

Supervisé par :

Mme. Terrier

T.IS-E2B

12 juin 2017

13 Code source

13.1 Vues

13.1.1 AboutView.cs

```
   /*
    Class           :   AboutView.cs
    Description     :   Gives general information about the project
    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Diagnostics;
using System.Windows.Forms;

namespace PrisonersDilemmaCA
{
    public partial class AboutView : Form
    {
        public AboutView()
        {
            InitializeComponent();
        }

        /// <summary>
        /// Close the form
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void btnClose_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        /// <summary>
        /// Open the wiki page
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void linkLabel1_LinkClicked(object sender, ↩
            LinkLabelLinkClickedEventArgs e)
        {
            Process.Start(lblWikiLink.Text);
        }

        /// <summary>
        /// Open the github page
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void lblGithubLink_LinkClicked(object sender, ↩
            LinkLabelLinkClickedEventArgs e)
        {
            Process.Start(lblGithubLink.Text);
        }
    }
}
```

13.1.2 GenerateHelpView.cs

```
   /*
    Class           :   GenerateHelpView.cs
    Description     :   Gives help on generating grid of cells
    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Windows.Forms;

namespace PrisonersDilemmaCA
{
    public partial class GenerateHelpView : Form
    {
```

```
15          public GenerateHelpView()
16          {
17              InitializeComponent();
18          }
19
20          /// <summary>
21          /// Closes the form
22          /// </summary>
23          /// <param name="sender"></param>
24          /// <param name="e"></param>
25          private void btnClose_Click(object sender, EventArgs e)
26          {
27              this.Close();
28          }
29      }
30  }
```

13.1.3 GenerateView.cs

```
1     /*
2      Class            :   GenerateView.cs
3      Description      :   Allows the user to generate grids of cells with various ↩
           strategies
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6  */
7
8  using System;
9  using System.Collections.Generic;
10 using System.ComponentModel;
11 using System.Drawing;
12 using System.Linq;
13 using System.Windows.Forms;
14
15 namespace PrisonersDilemmaCA
16 {
17     public partial class GenerateView : Form
18     {
19         public Grid currentGrid { get; set; }
20         public List<Strategy> strategies { get; set; }
21
22         int nbOfStrategies;
23         int heightOfComponents = 40;
24         int widthOfComponents = 150;
25         int spacing;
26         int formHeight;
27         int formWidth;
28         List<TrackBar> trackbars;
29         List<Label> trackbarLabels;
30         List<int> lastTrackbarValues;
31
32         public GenerateView()
33         {
34             InitializeComponent();
35         }
36
37         /// <summary>
38         /// Generates the GUI dynamially on load
39         /// </summary>
40         /// <param name="sender"></param>
41         /// <param name="e"></param>
42         private void GenerateView_Load(object sender, EventArgs e)
43         {
44             // Store the number of available strategies we have
45             nbOfStrategies = strategies.Count;
46
47             // Define some values to create our view dynamically
48             spacing = heightOfComponents;
49             formHeight = 0;
50             formWidth = 0;
51
52             // Initialize our list of trackbars
53             trackbars = new List<TrackBar>();
54             trackbarLabels = new List<Label>();
55             lastTrackbarValues = new List<int>();
56
57             // Generate the interface dynamically
58             for (int i = 1; i <= nbOfStrategies; i++)
59             {
60                 Label tmpLabel = new Label();
```

```
61                  int x = spacing;
62                  int y = i * (heightOfComponents / 3 + spacing);
63
64                  // Set the location of the label
65                  tmpLabel.Location = new Point(x, y);
66                  tmpLabel.Width = widthOfComponents;
67                  tmpLabel.Height = heightOfComponents;
68
69                  // Set the label font
70                  tmpLabel.Font = new Font(FontFamily.GenericSansSerif, 11);
71
72                  // Add the label content
73                  string strategyName = strategies[i - 1].ToString();
74                  tmpLabel.Text = strategyName;
75
76                  // Create a trackbar
77                  TrackBar tmpTrackbar = new TrackBar();
78                  tmpTrackbar.Location = new Point(x + tmpLabel.Width + spacing, y);
79                  tmpTrackbar.Size = new Size(lblTitle.Width / 2 - x / 2, ↩
                        heightOfComponents);
80                  tmpTrackbar.Anchor = (AnchorStyles.Right | AnchorStyles.Top | ↩
                        AnchorStyles.Left);
81
82                  // Set the trackbar's parameters
83                  tmpTrackbar.Minimum = 0;
84                  tmpTrackbar.Maximum = 100;
85                  tmpTrackbar.Value = 0;
86                  tmpTrackbar.TickFrequency = 10;
87
88                  // Add an event handler to automatically refresh the interface
89                  tmpTrackbar.ValueChanged += new EventHandler(UpdateValues);
90
91                  // Add the trackbar to the list
92                  trackbars.Add(tmpTrackbar);
93                  lastTrackbarValues.Add(tmpTrackbar.Value);
94
95                  // Add a label for each trackbar
96                  Label tmpTrackbarLabel = new Label();
97
98                  // Set the location of the label (next to the trackbar)
99                  tmpTrackbarLabel.Location = new Point(tmpTrackbar.Left + ↩
                        tmpTrackbar.Width + spacing, y);
100                 tmpTrackbarLabel.Width = widthOfComponents;
101                 tmpTrackbarLabel.Height = heightOfComponents;
102
103                 // Anchor it
104                 tmpTrackbarLabel.Anchor = (AnchorStyles.Right | AnchorStyles.Top);
105
106                 // Set the label font
107                 tmpTrackbarLabel.Font = new Font(FontFamily.GenericSansSerif, 12);
108
109                 // Add it to the list
110                 trackbarLabels.Add(tmpTrackbarLabel);
111
112                 // Add the components to the form
113                 this.Controls.Add(tmpLabel);
114                 this.Controls.Add(tmpTrackbar);
115                 this.Controls.Add(tmpTrackbarLabel);
116
117                 // Set the form width and height
118                 formHeight += heightOfComponents + spacing - 5;
119             }
120             formWidth = lblTitle.Width + spacing * 3;
121
122             // Set the form's dimensions
123             this.MinimumSize = new Size(formWidth, formHeight);
124
125             // Center the form on screen
126             Rectangle screenSize = Screen.PrimaryScreen.Bounds;
127             int newX = screenSize.Width / 2 - this.Width / 2;
128             int newY = screenSize.Height / 2 - this.Height / 2;
129
130             this.Left = newX;
131             this.Top = newY;
132
133             // Update the controls
134             UpdateValues(null, null);
135         }
136
137         /// <summary>
138         /// Refreshes the interface and prevents the user from inputing incorrect values
139         /// </summary>
140         /// <param name="sender"></param>
141         /// <param name="e"></param>
```

```
142         public void UpdateValues(object sender, EventArgs e)
143         {
144             // Get the current total percentage
145             int sum = trackbars.Sum(item => item.Value);
146 
147             // Set the new max value of the trackbars
148             for (int i = 0; i < trackbars.Count; i++)
149             {
150                 // If we are at 100 percent, prevent the user from incrementing even more
151                 if (sum > 100)
152                 {
153                     if (trackbars[i].Value > lastTrackbarValues[i])
154                     {
155                         // Restore from the last value
156                         trackbars[i].Value = lastTrackbarValues[i];
157                     }
158                 }
159                 // Store the last value
160                 lastTrackbarValues[i] = trackbars[i].Value;
161 
162                 // Refresh the percentage of each of the trackbar's label
163                 trackbarLabels[i].Text = String.Format("{0}%", trackbars[i].Value);
164             }
165 
166             // The percentage is equal to the sum of each of the trackbar's value
167             sum = (sum > 100) ? 100 : sum;
168             pbPercentage.Value = sum;
169             lblPercentage.Text = String.Format("{0}%", sum);
170 
171             // Enable the button if we have 100% progress
172             btnApply.Enabled = (sum >= 100) ? true : false;
173         }
174 
175         /// <summary>
176         /// Closes the form
177         /// </summary>
178         /// <param name="sender"></param>
179         /// <param name="e"></param>
180         private void btnCancel_Click(object sender, EventArgs e)
181         {
182             this.Close();
183         }
184 
185         /// <summary>
186         /// Generates a new board with random cells
187         /// </summary>
188         /// <param name="sender"></param>
189         /// <param name="e"></param>
190         private void btnApply_Click(object sender, EventArgs e)
191         {
192             // Create a new random number generator
193             Random rng = new Random();
194 
195             Dictionary<Strategy, int> stratAndPercent = new Dictionary<Strategy, int>();
196             List<Strategy> toRemove = new List<Strategy>();
197 
198             // Filter out the unused strategies
199             for (int i = 0; i < nbOfStrategies; i++)
200             {
201                 if (!(trackbars[i].Value <= 0))
202                 {
203                     // Store the percentage of the remaining strategies
204                     stratAndPercent.Add(strategies[i], trackbars[i].Value);
205                 }
206             }
207 
208             Grid tmpGrid = new Grid(currentGrid.Width, currentGrid.Height, ↩
                    currentGrid.NbLines, currentGrid.NbCols, currentGrid.PayoffMatrix);
209 
210             // Generate a new board
211             tmpGrid.generate(stratAndPercent);
212             currentGrid.Cells = tmpGrid.Cells;
213 
214             // Close the form
215             this.Close();
216         }
217 
218         /// <summary>
219         /// Open the help form
220         /// </summary>
221         /// <param name="sender"></param>
222         /// <param name="e"></param>
223         private void GenerateView_HelpButtonClicked(object sender, CancelEventArgs e)
224         {
```

```
225             GenerateHelpView helpView = new GenerateHelpView();
226 
227             if (helpView.ShowDialog() == DialogResult.OK)
228             {
229                 // User pressed the close button
230             }
231         }
232     }
233 }
```

13.1.4 MainView.cs

```
1    /*
2     Class           :   MainView.cs
3     Description     :   Main view of the application.
4     Author          :   SEEMULLER Julien
5     Date            :   10.04.2017
6 */
7 
8 using LiveCharts;
9 using LiveCharts.Wpf;
10 using System;
11 using System.Collections.Generic;
12 using System.Windows.Forms;
13 
14 namespace PrisonersDilemmaCA
15 {
16     public partial class MainView : Form
17     {
18 
19         /*************************************************************************
20          *                              GLOBAL VARIABLES                         *
21          *************************************************************************/
22         Grid mainGrid;
23         PayoffMatrix payoffMatrix;
24         List<Strategy> availableStrategies;
25         bool isClickingOnGrid = false;
26         bool isAutoplaying = false;
27         int mouseX = 0;
28         int mouseY = 0;
29         int generation = 0;
30         WrapMode gridWrappingMode = WrapMode.Default;
31 
32         const int MAX_NB_ELEMENTS_IN_CHART = 10;
33         const int DEFAULT_NORMAL_VIEW_WIDTH = 610;
34         const int DEFAULT_EXTENDED_VIEW_WIDTH = 1100;
35         /*************************************************************************
36          *                                EVENTS                                 *
37          *************************************************************************/
38         /// <summary>
39         /// Default constructor
40         /// </summary>
41         public MainView()
42         {
43             InitializeComponent();
44         }
45 
46         /// <summary>
47         /// Set up the form after it is loaded
48         /// </summary>
49         /// <param name="sender"></param>
50         /// <param name="e"></param>
51         private void MainView_Load(object sender, EventArgs e)
52         {
53             // Make a list of all our available strategies
54             availableStrategies = new List<Strategy>();
55 
56             // To add more strategies, add them to the list
57             availableStrategies.Add(new StratRandom());
58             availableStrategies.Add(new StratTitForTat());
59             availableStrategies.Add(new StratBlinker());
60             availableStrategies.Add(new StratAlwaysCooperate());
61             availableStrategies.Add(new StratAlwaysDefect());
62             availableStrategies.Add(new StratTitForTwoTats());
63             availableStrategies.Add(new StratGrimTrigger());
64             availableStrategies.Add(new StratFortress());
65             availableStrategies.Add(new StratAdaptativePavlov());
66             availableStrategies.Add(new StratSuspiciousTitForTat());
67 
68             // Sort the list
```

```
69              availableStrategies.Sort();
70
71              // Initialize the payoff matrix with default values
72              payoffMatrix = new PayoffMatrix();
73
74              // Initialize our grid of cells
75              mainGrid = new Grid(pbGrid.Width, pbGrid.Height, tbLines.Value, ↩
                    tbColumns.Value, payoffMatrix, gridWrappingMode);
76
77              // Initialise the combobox with strategies and colors
78              cbStrategies.AddStrategies(availableStrategies);
79
80              // Select the first element by default
81              cbStrategies.SelectedIndex = 0;
82
83              // Set the user help text
84              lblUserHelp.Text = "Click on a cell to change its strategy."
85                  + Environment.NewLine + "The default strategy is "
86                  + Cell.DEFAULT_STRATEGY.ToString();
87
88              // Update the other labels
89              updateLabels();
90
91              // CHARTS
92              // Pie chart
93              pieStrategy.InnerRadius = 50;
94              pieStrategy.LegendLocation = LegendLocation.Right;
95              pieStrategy.DisableAnimations = true;
96              pieStrategy.Series = new SeriesCollection();
97
98              foreach (Strategy strategy in availableStrategies)
99              {
100                 // Get the color from the strategy
101                 System.Windows.Media.BrushConverter converter = new ↩
                        System.Windows.Media.BrushConverter();
102                 System.Windows.Media.Brush brush = ↩
                        (System.Windows.Media.Brush)converter.ConvertFromString(strategy.getColor().ToHex()
103
104                 // Create an object for storing values on the pie chart
105                 PieSeries stratToAdd = new PieSeries
106                 {
107                     Title = strategy.ToString(),
108                     Values = new ChartValues<double>
109                         {
110                             mainGrid.findCountOfStrategy(strategy)
111                         },
112                     DataLabels = true,
113                     Fill = brush
114                 };
115
116                 stratToAdd.Visibility = System.Windows.Visibility.Hidden;
117
118
119                 // Add the values to the pie chart
120                 pieStrategy.Series.Add(stratToAdd);
121             }
122
123             // Cartesian
124             cartesianStrategy.LegendLocation = LegendLocation.Right;
125             cartesianStrategy.AxisX.Add(new Axis
126             {
127                 Title = "Current Generation",
128                 LabelFormatter = value => value.ToString()
129             });
130
131             cartesianStrategy.AxisY.Add(new Axis
132             {
133                 Title = "Number of Days in Prison",
134                 LabelFormatter = value => value.ToString(),
135
136             });
137
138             // Initialize the cartesian chart
139             initializeChart();
140             updateDonutChart();
141         }
142
143         /// <summary>
144         /// Force refresh the form each tick of the timer
145         /// Default tickrate : 16ms -> 60fps
146         /// </summary>
147         /// <param name="sender"></param>
148         /// <param name="e"></param>
149         private void MainTimer_Tick(object sender, EventArgs e)
```

```
150         {
151             Refresh();
152         }
153 
154         private void pbGrid_Paint(object sender, PaintEventArgs e)
155         {
156             // Draw code here
157             mainGrid.draw(e.Graphics);
158         }
159 
160         /// <summary>
161         /// Updates when changing the number of cells horizontally
162         /// </summary>
163         /// <param name="sender"></param>
164         /// <param name="e"></param>
165         private void trackBar1_Scroll(object sender, EventArgs e)
166         {
167             updateGrid();
168         }
169 
170         /// <summary>
171         /// Updates when changing the number of cells vertically
172         /// </summary>
173         /// <param name="sender"></param>
174         /// <param name="e"></param>
175         private void trackBar2_Scroll(object sender, EventArgs e)
176         {
177             updateGrid();
178         }
179 
180         /// <summary>
181         /// Open the generation form
182         /// </summary>
183         /// <param name="sender"></param>
184         /// <param name="e"></param>
185         private void generateNewBoardToolStripMenuItem_Click(object sender, EventArgs e)
186         {
187             interruptTimer();
188 
189             // Pass the grid and list of strategies to the form and open them
190             GenerateView generateView = new GenerateView();
191             generateView.currentGrid = this.mainGrid;
192             generateView.strategies = this.availableStrategies;
193 
194             if (generateView.ShowDialog() == DialogResult.OK)
195             {
196                 // The user has validated his input
197                 // Reset the generation count
198                 generation = 0;
199 
200                 // Update the GUI
201                 updateLabels();
202                 updateDonutChart();
203                 initializeChart();
204                 mainGrid.setColorMode(ColorMode.Strategy);
205             }
206         }
207 
208         /// <summary>
209         /// Open the payoff matrix parameters
210         /// </summary>
211         /// <param name="sender"></param>
212         /// <param name="e"></param>
213         private void payoffMatrixToolStripMenuItem_Click(object sender, EventArgs e)
214         {
215             interruptTimer();
216 
217             // Pass the PayoffMatrix object as parameter to the form and open it
218             PayoffMatrixView matrixView = new PayoffMatrixView();
219             matrixView.currentMatrix = this.payoffMatrix;
220 
221             if (matrixView.ShowDialog() == DialogResult.Yes)
222             {
223                 // The user has validated his input
224             }
225         }
226 
227         /// <summary>
228         /// Open the strategy benchmark window
229         /// </summary>
230         /// <param name="sender"></param>
231         /// <param name="e"></param>
232         private void benchmarkStrategiesToolStripMenuItem_Click(object sender, ↩
            EventArgs e)
```

```
233         {
234             interruptTimer();
235 
236             // Pass the PayoffMatrix object as parameter to the form and open it
237             BenchmarkView benchmarkView = new BenchmarkView();
238 
239             // Pass some values for the view
240             benchmarkView.strategies = availableStrategies;
241             benchmarkView.matrix = payoffMatrix;
242 
243             if (benchmarkView.ShowDialog() == DialogResult.OK)
244             {
245 
246             }
247         }
248 
249         /// <summary>
250         /// Open the about window
251         /// </summary>
252         /// <param name="sender"></param>
253         /// <param name="e"></param>
254         private void helpToolStripMenuItem_Click(object sender, EventArgs e)
255         {
256             interruptTimer();
257             AboutView view = new AboutView();
258 
259             if (view.ShowDialog() == DialogResult.OK)
260             {
261                 // The user has validated his input
262             }
263         }
264 
265         /// <summary>
266         /// Update a flag when we click on the grid
267         /// </summary>
268         /// <param name="sender"></param>
269         /// <param name="e"></param>
270         private void pbGrid_MouseDown(object sender, MouseEventArgs e)
271         {
272             isClickingOnGrid = true;
273             updateCellState();
274         }
275 
276         /// <summary>
277         /// Update a flag when we release our click on the grid
278         /// </summary>
279         /// <param name="sender"></param>
280         /// <param name="e"></param>
281         private void pbGrid_MouseUp(object sender, MouseEventArgs e)
282         {
283             isClickingOnGrid = false;
284             updateCellState();
285             updateDonutChart();
286         }
287 
288         /// <summary>
289         /// Updates the clicked cell with its new strategy
290         /// </summary>
291         /// <param name="sender"></param>
292         /// <param name="e"></param>
293         private void pbGrid_MouseMove(object sender, MouseEventArgs e)
294         {
295             mouseX = e.X;
296             mouseY = e.Y;
297             updateCellState();
298         }
299 
300         /// <summary>
301         ///
302         /// </summary>
303         /// <param name="sender"></param>
304         /// <param name="e"></param>
305         private void btnPlayPause_Click(object sender, EventArgs e)
306         {
307             // Change the button's text and launch the timer
308             switchPlayPauseState();
309         }
310 
311         /// <summary>
312         /// Manually steps forward (click)
313         /// </summary>
314         /// <param name="sender"></param>
315         /// <param name="e"></param>
316         private void btnStep_Click(object sender, EventArgs e)
```

```
317         {
318             stepForward();
319         }
320 
321         /// <summary>
322         /// Automatically steps forwards
323         /// </summary>
324         /// <param name="sender"></param>
325         /// <param name="e"></param>
326         private void StepTimer_Tick(object sender, EventArgs e)
327         {
328             stepForward();
329         }
330 
331         /// <summary>
332         /// Change the autostep speed
333         /// </summary>
334         /// <param name="sender"></param>
335         /// <param name="e"></param>
336         private void tbTimerSpeed_Scroll(object sender, EventArgs e)
337         {
338             StepTimer.Interval = tbTimerSpeed.Value;
339             updateLabels();
340         }
341 
342 
343         /// <summary>
344         /// Switch back to strategy color mode when we click on the strategy combo box
345         /// </summary>
346         /// <param name="sender"></param>
347         /// <param name="e"></param>
348         private void cbStrategies_Click(object sender, EventArgs e)
349         {
350             // Interrupt the autoplay if it is running
351             interruptTimer();
352             mainGrid.setColorMode(ColorMode.Strategy);
353             updateLabels();
354             Refresh();
355         }
356 
357         // Clears the board and fills it with the default cell
358         private void btnClear_Click(object sender, EventArgs e)
359         {
360             updateGrid();
361         }
362 
363 
364         /// <summary>
365         /// Alternates between normal and extended view
366         /// </summary>
367         /// <param name="sender"></param>
368         /// <param name="e"></param>
369         private void tsExtendedView_CheckedChanged(object sender, EventArgs e)
370         {
371             if (tsExtendedView.Checked)
372             {
373                 // Switch to extended view
374                 this.Width = DEFAULT_EXTENDED_VIEW_WIDTH;
375             }
376             else
377             {
378                 // Switch to normal view
379                 this.Width = DEFAULT_NORMAL_VIEW_WIDTH;
380             }
381         }
382 
383         /// <summary>
384         /// Alternates between default and torus wrapping mode
385         /// </summary>
386         /// <param name="sender"></param>
387         /// <param name="e"></param>
388         private void tsWrapMode_CheckedChanged(object sender, EventArgs e)
389         {
390             if (tsWrapMode.Checked)
391             {
392                 gridWrappingMode = WrapMode.Torus;
393             }
394             else
395             {
396                 gridWrappingMode = WrapMode.Default;
397             }
398 
399             // Reset the grid to regenerate the neighbors lists
400             updateGrid();
```

```
401         }
402
403         /*************************************************************************
404          *                                FUNCTIONS                              *
405          *************************************************************************/
406         /// <summary>
407         /// Switch the states between play and pause
408         /// </summary>
409         public void switchPlayPauseState()
410         {
411             if (isAutoplaying)
412             {
413                 btnPlayPause.Text = "4";
414                 StepTimer.Stop();
415             }
416             else
417             {
418                 btnPlayPause.Text = ";";
419                 StepTimer.Start();
420             }
421
422             // Invert the state
423             isAutoplaying = !isAutoplaying;
424         }
425
426
427         /// <summary>
428         /// Steps forward in time
429         /// </summary>
430         private void stepForward()
431         {
432             // Steps forward
433             mainGrid.step();
434             mainGrid.setColorMode(ColorMode.Playing);
435
436             // Increment the generation count
437             generation++;
438
439             // Update the GUI
440             updateLabels();
441             updateDonutChart();
442             addDataToChart();
443         }
444
445
446
447         /// <summary>
448         /// Pause the "autostep" timer if it is running
449         /// </summary>
450         public void interruptTimer()
451         {
452             if (isAutoplaying)
453             {
454                 switchPlayPauseState();
455             }
456         }
457
458         /// <summary>
459         /// Updates the labels with new information
460         /// </summary>
461         private void updateLabels()
462         {
463             // Trackbar labels
464             lblLines.Text = String.Format("Rows : {0}", tbLines.Value);
465             lblCols.Text = String.Format("Columns : {0}", tbColumns.Value);
466
467             // Grid label
468             lblGridInfo.Text = String.Format("{0}x{1} Grid - Mode : {2} - Generation ↩
                    {3}", tbLines.Value, tbColumns.Value, mainGrid.ColorMode.ToString(), ↩
                    generation);
469
470             // Speed labels
471             lblSpeedValue.Text = "automatically steps every " + tbTimerSpeed.Value + " ↩
                    [ms]";
472         }
473
474         /// <summary>
475         /// If the user is clicking on the grid, update the cell under the user's cursor
476         /// </summary>
477         public void updateCellState()
478         {
479             if (isClickingOnGrid)
480             {
```

```
481                 Strategy selectedStrategy = ↩
                        availableStrategies[cbStrategies.SelectedIndex];
482                 this.mainGrid.onClick(mouseX, mouseY, selectedStrategy);
483 
484                 // Interrupt the autoplay if it is running
485                 interruptTimer();
486 
487                 // Change the color mode
488                 mainGrid.setColorMode(ColorMode.Strategy);
489                 updateLabels();
490                 Refresh();
491             }
492         }
493 
494         /// <summary>
495         /// Updates the grid with new values (Re-create the grid)
496         /// </summary>
497         private void updateGrid()
498         {
499             // Interrupt the autoplay if it is running
500             interruptTimer();
501             mainGrid = new Grid(pbGrid.Width, pbGrid.Height, tbLines.Value, ↩
                    tbColumns.Value, payoffMatrix, gridWrappingMode);
502 
503             // Reset the generation count
504             generation = 0;
505 
506             // Update the labels and chart
507             updateLabels();
508             updateDonutChart();
509             initializeChart();
510         }
511 
512         /// <summary>
513         /// Updates the donut chart on the main view
514         /// </summary>
515         private void updateDonutChart()
516         {
517             // Update the donut chart
518             int count = 0;
519             foreach (Series serie in pieStrategy.Series)
520             {
521                 if (mainGrid.findCountOfStrategy(availableStrategies[count]) > 0)
522                 {
523                     serie.Visibility = System.Windows.Visibility.Visible;
524                     serie.Values = new ChartValues<double> { ↩
                            mainGrid.findCountOfStrategy(availableStrategies[count]) };
525                 }
526                 else
527                 {
528                     serie.Visibility = System.Windows.Visibility.Hidden;
529                 }
530 
531                 count++;
532             }
533         }
534 
535         /// <summary>
536         /// Initialize the cartesian chart with the base values
537         /// </summary>
538         public void initializeChart()
539         {
540             cartesianStrategy.Series = new SeriesCollection();
541             foreach (Strategy strategy in availableStrategies)
542             {
543                 // Get the color from the strategy
544                 System.Windows.Media.BrushConverter converter = new ↩
                        System.Windows.Media.BrushConverter();
545                 System.Windows.Media.Brush brush = ↩
                        (System.Windows.Media.Brush)converter.ConvertFromString(strategy.getColor().ToHex(60
546                 System.Windows.Media.Brush stroke = ↩
                        (System.Windows.Media.Brush)converter.ConvertFromString(strategy.getColor().ToHex()
547 
548                 // Create an object for storing values on the line chart
549                 LineSeries stratToAdd = new LineSeries
550                 {
551                     Title = strategy.ToString(),
552                     Values = new ChartValues<double> { 0 },
553                     PointGeometry = DefaultGeometries.None,
554                     PointGeometrySize = 15,
555                     Fill = brush,
556                     Stroke = stroke
557                 };
558 
```

```
559                 // Hide the unused strategies
560                 if (mainGrid.findCountOfStrategy(strategy) <= 0)
561                 {
562                     stratToAdd.Visibility = System.Windows.Visibility.Hidden;
563                 }
564 
565                 cartesianStrategy.AxisX[0].MinValue = 0;
566                 cartesianStrategy.AxisX[0].MaxValue = MAX_NB_ELEMENTS_IN_CHART;
567 
568                 // Add the values to the pie chart
569                 cartesianStrategy.Series.Add(stratToAdd);
570             }
571         }
572 
573         /// <summary>
574         /// Adds data to the cartesian chart
575         /// </summary>
576         private void addDataToChart()
577         {
578             int count = 0;
579 
580             // Readjust the X axis
581             cartesianStrategy.AxisX[0].MaxValue = generation;
582             if (generation > MAX_NB_ELEMENTS_IN_CHART)
583             {
584                 cartesianStrategy.AxisX[0].MinValue = generation - ↩
                        MAX_NB_ELEMENTS_IN_CHART;
585             }
586 
587             foreach (Series serie in cartesianStrategy.Series)
588             {
589 
590                 // Check the currently used strategies
591                 if (mainGrid.findCountOfStrategy(availableStrategies[count]) > 0)
592                 {
593                     // Add the average score of each used strategy
594                     serie.Values.Add(mainGrid.findAvgScoreOfStrategy(availableStrategies[count]));
595                     serie.Visibility = System.Windows.Visibility.Visible;
596                 }
597                 else
598                 {
599                     // Add 0 to the unused values (allows the graph to stay synced)
600                     serie.Values.Add((double)0);
601                     serie.Visibility = System.Windows.Visibility.Hidden;
602                 }
603 
604                 count++;
605             }
606         }
607 
608         /// <summary>
609         /// Save the current grid in a serialized format
610         /// </summary>
611         /// <param name="sender"></param>
612         /// <param name="e"></param>
613         private void saveGridToolStripMenuItem_Click(object sender, EventArgs e)
614         {
615             // Open a file dialog for the user to save the file
616             SaveFileDialog sfd = new SaveFileDialog();
617             sfd.Filter = "XML files|*.xml";
618 
619             if (sfd.ShowDialog() == DialogResult.OK)
620             {
621                 // Save the data to the path
622                 mainGrid.saveData(sfd.FileName);
623 
624                 // Notify the user
625                 MessageBox.Show("Grid exported successfully");
626             }
627         }
628 
629         /// <summary>
630         /// Load the current grid from a serialized file
631         /// </summary>
632         /// <param name="sender"></param>
633         /// <param name="e"></param>
634         private void loadGridToolStripMenuItem_Click(object sender, EventArgs e)
635         {
636             // Open a file dialog for the user to load the file
637             OpenFileDialog ofd = new OpenFileDialog();
638             ofd.Filter = "XML files|*.xml";
639 
640             if (ofd.ShowDialog() == DialogResult.OK)
641             {
```

```
642                 // Load the data from the path
643                 mainGrid.loadData(ofd.FileName);
644 
645                 // Update the trackbars manually
646                 tbLines.Value = mainGrid.NbLines;
647                 tbColumns.Value = mainGrid.NbCols;
648                 this.updateLabels();
649 
650                 // Notify the user
651                 MessageBox.Show("Grid loaded successfully");
652             }
653         }
654     }
655 }
```

13.1.5 PayoffMatrixHelpView.cs

```
1    /*
2     Class           :   PayoffMatrixHelpView.cs
3     Description     :   Gives help to the user on payoff matrixes
4     Author          :   SEEMULLER Julien
5     Date            :   10.04.2017
6 */
7 
8 using System;
9 using System.Windows.Forms;
10 
11 namespace PrisonersDilemmaCA
12 {
13     public partial class PayoffMatrixHelpView : Form
14     {
15         public PayoffMatrixHelpView()
16         {
17             InitializeComponent();
18         }
19 
20         /// <summary>
21         /// Quit the help form
22         /// </summary>
23         /// <param name="sender"></param>
24         /// <param name="e"></param>
25         private void btnOk_Click(object sender, EventArgs e)
26         {
27             this.Close();
28         }
29     }
30 }
```

13.1.6 PayoffMatrixView.cs

```
1    /*
2     Class           :   PayoffMatrixView.cs
3     Description     :   Allows the user to interract with the payoff matrix
4     Author          :   SEEMULLER Julien
5     Date            :   10.04.2017
6 */
7 
8 using System;
9 using System.ComponentModel;
10 using System.Windows.Forms;
11 
12 namespace PrisonersDilemmaCA
13 {
14     public partial class PayoffMatrixView : Form
15     {
16         public PayoffMatrix currentMatrix { get; set; }
17 
18         public PayoffMatrixView()
19         {
20             InitializeComponent();
21         }
22 
23         private void PayoffMatrixView_Load(object sender, EventArgs e)
24         {
25             // Initialize our textboxes with the matrix data
```

```
26              rtbReward.Text = currentMatrix.Reward.ToString();
27              rtbSucker.Text = currentMatrix.Sucker.ToString();
28              rtbTemptation.Text = currentMatrix.Temptation.ToString();
29              rtbPunishment.Text = currentMatrix.Punishment.ToString();
30          }
31
32          // Apply the changes and quit
33          private void btnOk_Click(object sender, EventArgs e)
34          {
35              // Store the contents of the textboxes as integers
36              int t = Convert.ToInt32(rtbTemptation.Text);
37              int r = Convert.ToInt32(rtbReward.Text);
38              int p = Convert.ToInt32(rtbPunishment.Text);
39              int s = Convert.ToInt32(rtbSucker.Text);
40
41              // Check for matrix validity
42              // T < R < P < S
43              if (!(PayoffMatrix.isValid(t, r, p, s)))
44              {
45                  // If it is not valid we abort and tell the user
46                  MessageBox.Show(
47                          "The selected matrix is not valid."
48                          + Environment.NewLine
49                          + "Rules :"
50                          + Environment.NewLine
51                          + "[Temptation < Reward < Punishment < Sucker]"
52                          + Environment.NewLine
53                          + "[2 * Reward < Temptation + Sucker]"
54                      );
55              }
56              else
57              {
58                  // Else, we apply the changes
59                  currentMatrix.Reward = r;
60                  currentMatrix.Sucker = s;
61                  currentMatrix.Temptation = t;
62                  currentMatrix.Punishment = p;
63
64                  // Close the form
65                  this.Close();
66              }
67          }
68
69          // Cancel and quit
70          private void btnCancel_Click(object sender, EventArgs e)
71          {
72              this.Close();
73          }
74
75          private void PayoffMatrixView_HelpButtonClicked(object sender, CancelEventArgs e)
76          {
77              PayoffMatrixHelpView helpForm = new PayoffMatrixHelpView();
78
79              if (helpForm.ShowDialog() == DialogResult.OK)
80              {
81                  // User clicked on ok
82              }
83          }
84
85      }
86  }
```

13.2 Classes

13.2.1 Cell.cs

```
   /*
    Class           :   Cell.cs
    Description     :   Main class, represents one "player" of the prisoner's dilemma
    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Xml;
using System.Xml.Schema;
using System.Xml.Serialization;

namespace PrisonersDilemmaCA
{
    public class Cell : IXmlSerializable
    {
        #region fields
        #region consts
        public static readonly Strategy DEFAULT_STRATEGY = new StratTitForTat();
        public const int DEFAULT_BORDER_WIDTH = 1;
        private const int DEFAULT_X = 0;
        private const int DEFAULT_Y = 0;
        #endregion

        private int _x;                         // X position in the grid (should be multiplied by width if used for graphics)
        private int _y;                         // Y position in the grid (should be multiplied by height if used for graphics)
        private int _width;                     // Width of the cell (dependent on the grid)
        private int _height;                    // Height of the cell (dependent on the grid)
        private int _score;                     // Represents the number of days in prison
        private Strategy _strategy;             // The strategy used by the cell (ex : Tit for Tat)
        private Color _color;                   // The current color of the cell
        private List<Cell> _neighbors;          // A list of references to the cells neighbors
        private PayoffMatrix _payoffMatrix; // The payoff matrix used by the cell
        private Move _choice;                     // What the cell intends to do this turn (ex : Defect)
        private Stack<Move> _history;           // Complete history of the cell's actions (ex : C, C, C, D, C, D, etc...)
        #endregion

        #region properties
        public int X
        {
            get { return _x; }
            set { _x = value; }
        }

        public int Y
        {
            get { return _y; }
            set { _y = value; }
        }

        public int Width
        {
            get { return _width; }
            set { _width = value; }
        }

        public int Height
        {
            get { return _height; }
            set { _height = value; }
        }

        public int Score
        {
            get { return _score; }
            set { _score = value; }
        }

        public Strategy Strategy
```

```
73          {
74              get { return _strategy; }
75              set
76              {
77                  // Make sure it is a new instance of the strategy
78                  _strategy = (Strategy)Activator.CreateInstance(value.GetType());
79                  // Set the color when we change the strategy
80                  this.Color = this.Strategy.getColor();
81              }
82          }
83
84          public PayoffMatrix PayoffMatrix
85          {
86              get { return _payoffMatrix; }
87              set { _payoffMatrix = value; }
88          }
89
90          public Color Color
91          {
92              get { return _color; }
93              set { _color = value; }
94          }
95
96          public List<Cell> Neighbors
97          {
98              get { return _neighbors; }
99              set { _neighbors = value; }
100         }
101
102         private Move Choice
103         {
104             get { return _choice; }
105             set { _choice = value; }
106         }
107
108         public Stack<Move> History
109         {
110             get { return _history; }
111             set { _history = value; }
112         }
113         #endregion
114
115         #region constructors
116         /// <summary>
117         /// Designated constructor
118         /// </summary>
119         /// <param name="x">X coordinate of the cell on the grid</param>
120         /// <param name="y">Y coordinate of the cell on the grid</param>
121         /// <param name="strategy">Current strategy of the cell</param>
122         /// <param name="matrix">Payoff matrix used to determine the score of each ↩
                cell</param>
123         public Cell(int x, int y, Strategy strategy, PayoffMatrix matrix)
124         {
125             this.X = x;
126             this.Y = y;
127             this.Strategy = strategy;
128             this.PayoffMatrix = matrix;
129             this.Score = 0;
130
131             this.Neighbors = new List<Cell>();
132             this.History = new Stack<Move>();
133
134             // Get the color of the cell from the current strategy
135             this.setColorFromStrategy();
136
137             // Starts with a move relevent to the strategy
138             this.chooseNextMove();
139         }
140
141         /// <summary>
142         /// Conveniance constructor
143         /// </summary>
144         /// <param name="x"></param>
145         /// <param name="y"></param>
146         public Cell(int x, int y, PayoffMatrix matrix)
147             : this(x, y, DEFAULT_STRATEGY, matrix)
148         {
149             // No code
150         }
151
152         /// <summary>
153         /// Default constructor
154         /// </summary>
155         public Cell()
```

```
156             : this(DEFAULT_X, DEFAULT_Y, new PayoffMatrix())
157         {
158
159         }
160         #endregion
161
162         #region methods
163         /// <summary>
164         /// Plays a game of the prisoners dilemma with the cell's neighbors using the ↩
                cell's current strategy
165         /// </summary>
166         public void step()
167         {
168             // Go and play with each of our neighbors
169             List<int> scores = new List<int>();
170             foreach (Cell neighbor in this.Neighbors)
171             {
172                 // Play a game and store the result
173                 scores.Add(PayoffMatrix.returnPayoff(this.Choice, neighbor.Choice));
174             }
175
176             // We get the best score of the cell
177             this.Score = scores.Min();
178
179             // Update the color of the cell
180             this.setColorFromMove();
181         }
182
183         /// <summary>
184         /// Choose the next move using our strategy and neighbors
185         /// </summary>
186         public void chooseNextMove()
187         {
188             this.Choice = this.Strategy.chooseMove(this, this.Neighbors);
189         }
190
191         /// <summary>
192         /// Updates the last move of the cell
193         /// </summary>
194         public void updateLastMove()
195         {
196             this.History.Push(this.Choice);
197         }
198
199         /// <summary>
200         /// Function used to draw the cell
201         /// </summary>
202         /// <param name="g">The graphical element we use to draw</param>
203         public void draw(Graphics g)
204         {
205             // Color of the cell
206             SolidBrush cellColor = new SolidBrush(this.Color);
207
208             // Border parameters (color, width)
209             Pen borderColor = new Pen(Color.Black, DEFAULT_BORDER_WIDTH);
210
211             // Draw the cell
212             g.FillRectangle(cellColor, this); // Implicitly converted as a rectangle
213             g.DrawRectangle(borderColor, this);
214         }
215
216
217         /// <summary>
218         /// Implicit conversion to rectangle to simplify other functions
219         /// </summary>
220         /// <param name="cell">The cell used for conversion</param>
221         /// <returns></returns>
222         public static implicit operator Rectangle(Cell cell)
223         {
224             return new Rectangle(cell.X * cell.Width, cell.Y * cell.Height, ↩
                    cell.Width, cell.Height);
225         }
226
227         /// <summary>
228         /// On click, we update the cell's strategy with a new one
229         /// </summary>
230         /// <param name="x">The x coordinate in pixels</param>
231         /// <param name="y">The y coordinate in pixels</param>
232         public void onClick(int x, int y, Strategy strat)
233         {
234             Rectangle hitbox = this;
235
236             // If we are the cell that is hit, update our strategy and clear it's history
237             if (hitbox.Contains(x, y))
```

```
238             {
239                 updateStrategy(strat);
240             }
241         }
242 
243         /// <summary>
244         /// Updates the strategy of the cell
245         /// </summary>
246         /// <param name="strat">The strategy to update the cell with</param>
247         public void updateStrategy(Strategy strat)
248         {
249             // Change the strategy
250             this.Strategy = strat;
251 
252             // Updates the cell's move with the new strategy
253             this.History.Clear();
254 
255             // We play a game with our neighbors to sync with the current game
256             this.chooseNextMove();
257             this.updateLastMove();
258             this.step();
259         }
260 
261         /// <summary>
262         /// Set the color of the cell according to its next move
263         /// </summary>
264         public void setColorFromMove()
265         {
266             switch (this.Choice)
267             {
268                 case Move.Cooperate:
269                     if (this.History.First() == Move.Defect)
270                     {
271                         this.Color = Color.FromArgb(230, 126, 34); // ORANGE
272                     }
273                     else
274                     {
275                         this.Color = Color.FromArgb(46, 204, 113); // GREEN
276                     }
277                     break;
278 
279 
280                 case Move.Defect:
281                     if (this.History.First() == Move.Cooperate)
282                     {
283                         this.Color = Color.FromArgb(241, 196, 15); // YELLOW
284                     }
285                     else
286                     {
287                         this.Color = Color.FromArgb(192, 57, 43); // RED
288                     }
289 
290                     break;
291             }
292         }
293 
294         /// <summary>
295         /// Set the color of the cell according to its strategy
296         /// </summary>
297         public void setColorFromStrategy()
298         {
299             this.Color = this.Strategy.getColor();
300         }
301 
302 
303 
304 
305 
306         //****************************//
307         // INTERFACE IXMLSERIALIZABLE //
308         //****************************//
309 
310         /// <summary>
311         /// Unused, see MSDN documentation :
312         /// "This method is reserved and should not be used. It should always return a ↩
                null value"
313         /// </summary>
314         /// <returns></returns>
315         public XmlSchema GetSchema()
316         {
317             return null;
318         }
319 
320 
```

```
321         /// <summary>
322         /// Reads through a serialized XML file to get the values for a cell
323         /// </summary>
324         /// <param name="reader">The XML reader attached to the serialized file</param>
325         public void ReadXml(XmlReader reader)
326         {
327 
328             int R = -1;
329             int G = -1;
330             int B = -1;
331 
332             reader.Read(); // Skip the beggining tab
333             if (reader.Name == "X")
334             {
335                 reader.Read(); // Read past the name tag
336                 this.X = int.Parse(reader.Value);
337                 reader.Read(); // Read past the value
338 
339             }
340             reader.Read(); // Read past the closing tag
341 
342             // repeat this process for every value...
343 
344             if (reader.Name == "Y")
345             {
346                 reader.Read();
347                 this.Y = int.Parse(reader.Value);
348                 reader.Read();
349             }
350             reader.Read();
351 
352             if (reader.Name == "Width")
353             {
354                 reader.Read();
355                 this.Width = int.Parse(reader.Value);
356                 reader.Read();
357             }
358             reader.Read();
359 
360             if (reader.Name == "Height")
361             {
362                 reader.Read();
363                 this.Height = int.Parse(reader.Value);
364                 reader.Read();
365             }
366             reader.Read();
367 
368             if (reader.Name == "Strategy")
369             {
370                 reader.Read();
371                 // Create a new instance of the strategy
372                 Type elementType = Type.GetType(reader.Value);
373                 this.Strategy = (Strategy)Activator.CreateInstance(elementType);
374                 reader.Read();
375             }
376             reader.Read();
377 
378 
379             if (reader.Name == "R")
380             {
381                 reader.Read();
382                 R = int.Parse(reader.Value);
383                 reader.Read();
384             }
385             reader.Read();
386 
387             if (reader.Name == "G")
388             {
389                 reader.Read();
390                 G = int.Parse(reader.Value);
391                 reader.Read();
392             }
393             reader.Read();
394 
395             if (reader.Name == "B")
396             {
397                 reader.Read();
398                 B = int.Parse(reader.Value);
399                 reader.Read();
400             }
401             reader.Read();
402 
403             // Check if the RGB values are assigned
404             if (R > 0 && G > 0 && B > 0)
```

```
405             {
406                 // Create a color
407                 this.Color = Color.FromArgb(R, G, B);
408 
409                 // Reset the color
410                 R = -1;
411                 G = -1;
412                 B = -1;
413             }
414 
415             if (reader.Name == "Score")
416             {
417                 reader.Read();
418                 // Tries to parse the reader value as a "Move" enum
419                 this.Score = int.Parse(reader.Value);
420                 reader.Read();
421             }
422             reader.Read();
423             reader.Read(); // Skip ending tag
424         }
425 
426 
427         /// <summary>
428         /// Write the cell's value to a serialized XML file
429         /// </summary>
430         /// <param name="writer">The XML reader attached to the serialized file</param>
431         public void WriteXml(XmlWriter writer)
432         {
433             // Set color from strategy before continuing
434             setColorFromStrategy();
435 
436             // Write the content of the cell to xml format
437             writer.WriteStartElement("X");
438             writer.WriteString(this.X.ToString());
439             writer.WriteEndElement();
440 
441             writer.WriteStartElement("Y");
442             writer.WriteString(this.Y.ToString());
443             writer.WriteEndElement();
444 
445             writer.WriteStartElement("Width");
446             writer.WriteString(this.Width.ToString());
447             writer.WriteEndElement();
448 
449             writer.WriteStartElement("Height");
450             writer.WriteString(this.Height.ToString());
451             writer.WriteEndElement();
452 
453             writer.WriteStartElement("Strategy");
454             writer.WriteString(this.Strategy.GetType().ToString());
455             writer.WriteEndElement();
456 
457             writer.WriteStartElement("R");
458             writer.WriteString(this.Color.R.ToString());
459             writer.WriteEndElement();
460 
461             writer.WriteStartElement("G");
462             writer.WriteString(this.Color.G.ToString());
463             writer.WriteEndElement();
464 
465             writer.WriteStartElement("B");
466             writer.WriteString(this.Color.B.ToString());
467             writer.WriteEndElement();
468 
469             writer.WriteStartElement("Score");
470             writer.WriteString(this.Score.ToString());
471             writer.WriteEndElement();
472 
473             /* HISTORY - UNUSED, INCREASED SIZE OF FILE EXPONENTIALLY WITH EACH ↩
                GENERATION
474             writer.WriteStartElement("History");
475             foreach (Move choice in this.History)
476             {
477                 writer.WriteStartElement("Choice");
478                 writer.WriteString(choice.ToString());
479                 writer.WriteEndElement();
480             }
481             writer.WriteEndElement();
482             */
483         }
484         #endregion
485     }
486 }
```

13.2.2 Grid.cs

```
1   /*
2    Class           :   Grid.cs
3    Description     :   Stores the cells of the cellular automaton,
4                        main model of the cellular automaton
5    Author          :   SEEMULLER Julien
6    Date            :   10.04.2017
7   */
8
9   using System;
10  using System.Collections.Generic;
11  using System.Drawing;
12  using System.IO;
13  using System.Linq;
14  using System.Text;
15  using System.Threading.Tasks;
16  using System.Xml.Serialization;
17
18  namespace PrisonersDilemmaCA
19  {
20      public class Grid
21      {
22          #region fields
23
24          #region consts
25          public const int NEAREST_NEIGHBOR_RANGE = 1;    // Change the "radius" at which we consider cells neighbors
26          private const int DEFAULT_HEIGHT = 100;
27          private const int DEFAULT_WIDTH = 100;
28          private const int DEFAULT_NB_COLS = 10;
29          private const int DEFAULT_NB_LINES = 10;
30          private const string DEFAULT_DATA_FILEPATH = "xml/grid.xml";
31
32          public const WrapMode DEFAULT_WRAP_MODE = WrapMode.Torus;
33          #endregion
34
35          private Cell[,] _cells;                         // 2D array containing the cells
36          private int _width;                             // Width of the grid in pixels
37          private int _height;                            // Height of the grid in pixels
38          private int _nbLines;                           // Number of lines in the grid (y)
39          private int _nbCols;                            // Number of columns in the grid (x)
40          private PayoffMatrix _payoffMatrix;             // Payoff matrix to be distributed to cells
41          private ColorMode _colorMode;                   // The current color mode of the grid (cf. ColorMode enum)
42          private WrapMode _wrapMode;                     // The current wrapping mode of the grid (cf. WrapMode enum)
43          private List<Cell> _serializableCells;          // Since [,] is not serializable, we make a list of cell before serializing.
44          #endregion
45
46          #region properties
47          [XmlIgnore]
48          public Cell[,] Cells
49          {
50              get { return _cells; }
51              set { _cells = value; }
52          }
53
54          public List<Cell> SerializableCells
55          {
56              get { return _serializableCells; }
57              set { _serializableCells = value; }
58          }
59
60          public int Width
61          {
62              get { return _width; }
63              set { _width = value; }
64          }
65
66          public int Height
67          {
68              get { return _height; }
69              set { _height = value; }
70          }
71
72          public int NbCols
73          {
74              get { return _nbCols; }
```

```
75              set { _nbCols = value; }
76          }
77
78          public int NbLines
79          {
80              get { return _nbLines; }
81              set { _nbLines = value; }
82          }
83
84          public PayoffMatrix PayoffMatrix
85          {
86              get { return _payoffMatrix; }
87              set { _payoffMatrix = value; }
88          }
89
90          public ColorMode ColorMode
91          {
92              get { return _colorMode; }
93              set { _colorMode = value; }
94          }
95
96          public WrapMode WrapMode
97          {
98              get { return _wrapMode; }
99              set { _wrapMode = value; }
100         }
101         #endregion
102
103         #region constructors
104         /// <summary>
105         /// Designated constructor
106         /// </summary>
107         /// <param name="width">The width of the grid in pixels</param>
108         /// <param name="height">The height of the grid in pixels</param>
109         /// <param name="nbCols">The number of columns of the grid</param>
110         /// <param name="nbLines">The number of lines of the grid</param>
111         public Grid(int width, int height, int nbLines, int nbCols, PayoffMatrix ↩
                matrix, WrapMode wrapmode, Strategy strategy)
112         {
113             this.Width = width;
114             this.Height = height;
115             this.NbLines = nbLines;
116             this.NbCols = nbCols;
117             this.PayoffMatrix = matrix;
118             this.ColorMode = ColorMode.Strategy;
119             this.WrapMode = wrapmode;
120
121             // Initialize our list of cells
122             this.Cells = new Cell[nbLines, nbCols];
123
124             // Calculate the width and the height of a cell
125             int cellWidth = this.Width / nbCols;
126             int cellHeight = this.Height / nbLines;
127
128             // Go through each possible slot in the grid
129             for (int y = 0; y < this.NbLines; y++)
130             {
131                 for (int x = 0; x < this.NbCols; x++)
132                 {
133                     // Create a temporary cell with the default strategy
134                     Cell tmpCell = new Cell(x, y, strategy, this.PayoffMatrix);
135
136                     // Set the cell's height according to the grid's need
137                     tmpCell.Width = cellWidth;
138                     tmpCell.Height = cellHeight;
139
140                     // Add the cell to the list
141                     this.Cells[y, x] = tmpCell;
142                 }
143             }
144
145             foreach (Cell cell in this.Cells)
146             {
147                 // Make each cell aware of its neighbors
148                 cell.Neighbors = findCellNeighbors(cell);
149             }
150         }
151
152         /// <summary>
153         /// Conveniance constructor
154         /// </summary>
155         public Grid(int width, int height, int nbLines, int nbCols, PayoffMatrix ↩
                matrix, WrapMode wrapmode)
```

```
156             : this(width, height, nbLines, nbCols, matrix, wrapmode, ↩
                    Cell.DEFAULT_STRATEGY)
157         {
158             // No code
159         }
160
161         /// <summary>
162         /// Conveniance constructor 2
163         /// </summary>
164         public Grid(int width, int height, int nbLines, int nbCols, PayoffMatrix matrix)
165             : this(width, height, nbLines, nbCols, matrix, DEFAULT_WRAP_MODE, ↩
                    Cell.DEFAULT_STRATEGY)
166         {
167             // No code
168         }
169
170         /// <summary>
171         /// Conveniance constructor 3
172         /// </summary>
173         public Grid(int width, int height, int nbLines, int nbCols)
174             : this(width, height, nbLines, nbCols, new PayoffMatrix(), ↩
                    DEFAULT_WRAP_MODE, Cell.DEFAULT_STRATEGY)
175         {
176             // No code
177         }
178
179         /// <summary>
180         /// Default constructor
181         /// (Required for serialization)
182         /// </summary>
183         public Grid()
184             : this(DEFAULT_WIDTH, DEFAULT_HEIGHT, DEFAULT_NB_LINES, DEFAULT_NB_COLS, ↩
                    new PayoffMatrix())
185         {
186             // No code
187         }
188         #endregion
189
190         #region methods
191
192         /// <summary>
193         /// Steps forward in time
194         /// </summary>
195         public void step()
196         {
197             // Store each of the cell's last move
198             foreach (Cell cell in this.Cells)
199             {
200                 cell.updateLastMove();
201             }
202
203             // Choose each of the cell's next move
204             foreach (Cell cell in this.Cells)
205             {
206                 cell.chooseNextMove();
207             }
208
209             // Step forward (play the game)
210             foreach (Cell cell in this.Cells)
211             {
212                 cell.step();
213             }
214         }
215
216         /// <summary>
217         /// Draw every cell on the board and the grid around them
218         /// </summary>
219         /// <param name="g">The graphics element we draw on</param>
220         public void draw(Graphics g)
221         {
222             // Draw each cell
223             foreach (Cell cell in this.Cells)
224             {
225                 cell.draw(g);
226             }
227
228             // Avoid drawing errors due to rounding
229             Pen borderColor = new Pen(Color.Black, Cell.DEFAULT_BORDER_WIDTH * 2);
230             g.DrawLine(borderColor, 0, this.Height, this.Width, this.Height);
231             g.DrawLine(borderColor, this.Width, 0, this.Width, this.Height);
232         }
233
234         /// <summary>
235         /// Generates a board of cell from a dictionary of strategy and percentages
```

```
236         /// </summary>
237         /// <param name="strategyAndPercentages">Dictionary countaining the strategies ↩
                and their percentage of appearence</param>
238         public void generate(Dictionary<Strategy, int> strategyAndPercentages)
239         {
240             // Create a new random number generator
241             Random rng = new Random();
242 
243             // Create a list of a hundred elements representing the repartition of ↩
                    strategies
244             List<Strategy> strategyPopulation = new List<Strategy>();
245 
246             // Go through each possible strategy and percentage
247             foreach (var strat in strategyAndPercentages)
248             {
249                 // Fill the list with the current strategy the same number of times as ↩
                        the percentage
250                 for (int i = 0; i < strat.Value; i++)
251                 {
252                     strategyPopulation.Add(strat.Key);
253                 }
254             }
255 
256             // Go through each cell in the grid
257             foreach (Cell cell in this.Cells)
258             {
259                 // Choose a random strategy in the list and apply it to the current cell
260                 int rnd = rng.Next(strategyPopulation.Count);
261                 cell.updateStrategy(strategyPopulation[rnd]);
262             }
263         }
264 
265         /// <summary>
266         /// Generates a board of cell from a list of strategy and percentages
267         /// </summary>
268         /// <param name="strats">List of strategies</param>
269         /// <param name="percentages">List of percentages</param>
270         public void generate(List<Strategy> strats, List<int> percentages)
271         {
272             // Fill a dictionary with strategies and percentages
273             Dictionary<Strategy, int> stratAndPercentage = new Dictionary<Strategy, ↩
                    int>();
274 
275             int counter = 0;
276             foreach (var strategy in strats)
277             {
278                 stratAndPercentage.Add(strategy, percentages[counter]);
279             }
280 
281             // Generate the board
282             this.generate(stratAndPercentage);
283         }
284 
285         /// <summary>
286         /// Gets the cell at the given position in a toroidal fashion
287         /// </summary>
288         /// <param name="x">The x coordinate of the cell (on the board)</param>
289         /// <param name="y">The y coordinate of the cell (on the board)</param>
290         /// <returns></returns>
291         public Cell getCell(int x, int y)
292         {
293             // Find the corrisponding point in a toroidal fashion if we go out of bounds
294             Point point = getPointClampedInGrid(x, y);
295             int newX = point.X;
296             int newY = point.Y;
297 
298             // Return the correct cell
299             return this.Cells[newY, newX];
300         }
301 
302         /// <summary>
303         /// Gets a point and wraps around in a toroidal fashion if the point is out of ↩
                bounds.
304         /// The coordinates are in grid format (see nbLines, nbCols)
305         /// </summary>
306         /// <param name="x">The x coordinate of a point on the grid</param>
307         /// <param name="y">The y coordinate of a point on the grid</param>
308         /// <returns></returns>
309         public Point getPointClampedInGrid(int x, int y)
310         {
311             int newX = x;
312             int newY = y;
313 
314             // Check if we are out of bounds width-wise
```

```
315             if (newX >= this.NbCols)
316             {
317                 // ex : 20 -> 20 - width
318                 newX = newX - Convert.ToInt32(this.NbCols);
319             }
320 
321             if (newX < 0)
322             {
323                 // ex : -2 -> width - 2
324                 newX = Convert.ToInt32(this.NbCols) + newX;
325             }
326 
327             // Check if we are out of bounds height-wise
328             if (newY >= this.NbLines)
329             {
330                 // ex : 20 -> 20 - height
331                 newY = newY - Convert.ToInt32(this.NbLines);
332             }
333 
334             if (newY < 0)
335             {
336                 // ex : -2 -> height - 2
337                 newY = Convert.ToInt32(this.NbLines) + newY;
338             }
339 
340             return new Point(newX, newY);
341         }
342 
343 
344         /// <summary>
345         /// Find the current cell's nearest neighbors (default 8 per cell)
346         /// </summary>
347         /// <param name="cell">The cell used to search for neighbors</param>
348         /// <returns></returns>
349         public List<Cell> findCellNeighbors(Cell cell)
350         {
351             List<Cell> neighbors = new List<Cell>();
352 
353             // Go all around the cell to find its neighbors
354             for (int y = cell.Y - NEAREST_NEIGHBOR_RANGE; y <= cell.Y + ↩
                NEAREST_NEIGHBOR_RANGE; y++)
355             {
356                 for (int x = cell.X - NEAREST_NEIGHBOR_RANGE; x <= cell.X + ↩
                    NEAREST_NEIGHBOR_RANGE; x++)
357                 {
358                     // Avoid our own cell
359                     if (!((x == cell.X) && (y == cell.Y)))
360                     {
361                         // Add the neighbor depending on the mode
362                         switch (this.WrapMode)
363                         {
364                             case WrapMode.Default:
365                                 // In default mode, check if we are inside the grid
366                                 if ((x >= 0) && (y >= 0) && (x < this.NbCols) && (y < ↩
                                    this.NbLines))
367                                 {
368                                     neighbors.Add(this.getCell(x, y));
369                                 }
370                                 break;
371 
372                             case WrapMode.Torus:
373                                 neighbors.Add(this.getCell(x, y));
374                                 break;
375                         }
376 
377                     }
378                 }
379             }
380 
381             return neighbors;
382         }
383 
384 
385         /// <summary>
386         /// Update the strategy of the cell that has been hit by the cursor
387         /// </summary>
388         /// <param name="x">The x coordinate in pixels</param>
389         /// <param name="y">The y coordinate in pixels</param>
390         /// <param name="strat">The strategy to apply to the cell if it is hit</param>
391         public void onClick(int x, int y, Strategy strat)
392         {
393             foreach (Cell cell in this.Cells)
394             {
395                 cell.onClick(x, y, strat);
```

```
396             }
397         }
398 
399         /// <summary>
400         /// Sets a strategy for a cell according to grid coordinates
401         /// </summary>
402         /// <param name="x"></param>
403         /// <param name="y"></param>
404         /// <param name="strat"></param>
405         public void setStrategy(int x, int y, Strategy strat)
406         {
407             this.Cells[y, x].updateStrategy(strat);
408         }
409 
410         /// <summary>
411         /// Set the color depending on the mode
412         ///
413         /// When in strategy color mode  :  The color of the strategy is shown.
414         /// When in move color mode      :  The color of the last move is shown.
415         ///
416         /// </summary>
417         /// <param name="mode">The color mode to use</param>
418         public void setColorMode(ColorMode mode)
419         {
420             // Switch according to the mode
421             switch (mode)
422             {
423                 case ColorMode.Strategy:
424                     this.setColorFromStrategy();
425                     break;
426                 case ColorMode.Playing:
427                     this.setColorFromMove();
428                     break;
429             }
430 
431             this.ColorMode = mode;
432         }
433 
434         /// <summary>
435         /// Sets the cell's colors from thier strategy
436         /// </summary>
437         private void setColorFromStrategy()
438         {
439             foreach (Cell cell in this.Cells)
440             {
441                 cell.setColorFromStrategy();
442             }
443         }
444 
445         /// <summary>
446         /// Sets the cell's colors from thier last move
447         /// </summary>
448         private void setColorFromMove()
449         {
450             foreach (Cell cell in this.Cells)
451             {
452                 cell.setColorFromMove();
453             }
454         }
455 
456 
457         /// <summary>
458         /// Finds the number of times the given strategy appears on the board
459         /// </summary>
460         /// <param name="strategy">The strategy to look for</param>
461         /// <returns></returns>
462         public int findCountOfStrategy(Strategy strategy)
463         {
464             int count = 0;
465 
466             foreach (Cell cell in this.Cells)
467             {
468                 // Find every cell that has the same type as the current strategy
469                 if (strategy.GetType() == cell.Strategy.GetType())
470                 {
471                     count++;
472                 }
473             }
474 
475             // Return the result rounded down to two decimal places
476             return count;
477         }
478 
479         /// <summary>
```

```
480         /// Returns the average score of a strategy on the board
481         /// </summary>
482         /// <param name="strategy">The strategy to look for</param>
483         /// <returns></returns>
484         public double findAvgScoreOfStrategy(Strategy strategy)
485         {
486             double count = 0;
487             int i = 0;
488 
489             foreach (Cell cell in this.Cells)
490             {
491                 // Find every cell that has the same type as the current strategy
492                 if (strategy.GetType() == cell.Strategy.GetType())
493                 {
494                     // Increment the total score and the count
495                     count += cell.Score;
496                     i++;
497                 }
498             }
499 
500             // Find the percentage from the count
501             count = (count / i);
502 
503             // Return the result rounded down to two decimal places
504             return Math.Round(count, 2);
505         }
506 
507 
508         /// <summary>
509         /// Serializes and saves grid data to a path
510         /// </summary>
511         /// <param name="path">Where to save the file on the user's disk</param>
512         public void saveData(string path)
513         {
514             this.SerializableCells = this.Cells.asList();
515             FileStream fs = new FileStream(path, FileMode.Create);
516             XmlSerializer xs = new XmlSerializer(typeof(Grid));
517             xs.Serialize(fs, this);
518             fs.Close();
519         }
520 
521 
522         /// <summary>
523         /// Serialize and saves grid data to the default location
524         /// </summary>
525         public void saveData()
526         {
527             this.saveData(DEFAULT_DATA_FILEPATH);
528         }
529 
530 
531         /// <summary>
532         /// Load serialized data from a path
533         /// </summary>
534         /// <param name="path">Where to load the file on the user's disk</param>
535         public void loadData(string path)
536         {
537             Grid newGrid;
538 
539             XmlSerializer xs = new XmlSerializer(typeof(Grid));
540             using (StreamReader rd = new StreamReader(path))
541             {
542                 newGrid = xs.Deserialize(rd) as Grid;
543             }
544 
545             // rebuild the neighbors
546             newGrid.Cells = newGrid.SerializableCells.asArrayOfArray(newGrid.NbLines, ↩
                    newGrid.NbCols);
547             foreach (var cell in newGrid.Cells)
548             {
549                 cell.Neighbors = newGrid.findCellNeighbors(cell);
550             }
551 
552             // Set each of the values from the serialized data
553             this.Width = newGrid.Width;
554             this.Height = newGrid.Height;
555             this.NbCols = newGrid.NbCols;
556             this.NbLines = newGrid.NbLines;
557             this.Cells = newGrid.Cells;
558             this.PayoffMatrix = newGrid.PayoffMatrix;
559             this.WrapMode = newGrid.WrapMode;
560         }
561 
562         /// <summary>
```

```
563         /// Loads the serialized data from the default location
564         /// </summary>
565         public void loadData()
566         {
567             this.loadData(DEFAULT_DATA_FILEPATH);
568         }
569         #endregion
570     }
571 }
```

13.2.3 PayoffMatrix.cs

```
1    /*
2     Class           :   PayoffMatrix.cs
3     Description     :   Class used to modelize the prisoner's dilemma payoff matrix
4     Author          :   SEEMULLER Julien
5     Date            :   10.04.2017
6 */
7
8 using System;
9 using System.Collections.Generic;
10 using System.Linq;
11 using System.Text;
12 using System.Threading.Tasks;
13
14 namespace PrisonersDilemmaCA
15 {
16     public class PayoffMatrix
17     {
18         #region fields
19         #region consts
20         private const int DEFAULT_TEMPTATION_PAYOFF = 0;
21         private const int DEFAULT_REWARD_PAYOFF = 1;
22         private const int DEFAULT_PUNISHMENT_PAYOFF = 3;
23         private const int DEFAULT_SUCKER_PAYOFF = 5;
24         #endregion
25
26         private int _reward;        // Reward payoff
27         private int _sucker;        // Sucker's payoff
28         private int _temptation;    // Tempatation payoff
29         private int _punishment;    // Punishment payoff
30         #endregion
31
32         #region properties
33         public int Reward
34         {
35             get { return _reward; }
36             set { _reward = value; }
37         }
38
39         public int Sucker
40         {
41             get { return _sucker; }
42             set { _sucker = value; }
43         }
44
45         public int Temptation
46         {
47             get { return _temptation; }
48             set { _temptation = value; }
49         }
50
51         public int Punishment
52         {
53             get { return _punishment; }
54             set { _punishment = value; }
55         }
56         #endregion
57
58         #region constructors
59         /// <summary>
60         /// Designated constructor
61         ///
62         /// Rules :
63         /// T better than R better than P better than S
64         /// </summary>
65         /// <param name="t">Temptation payoff</param>
66         /// <param name="r">Reward payoff</param>
67         /// <param name="p">Punishment payoff</param>
68         /// <param name="s">Sucker's payoff</param>
```

```
69          public PayoffMatrix(int t, int r, int p, int s)
70          {
71              this.Temptation = t;
72              this.Reward = r;
73              this.Punishment = p;
74              this.Sucker = s;
75          }
76
77          /// <summary>
78          /// Default constructor
79          /// </summary>
80          public PayoffMatrix() : this(DEFAULT_TEMPTATION_PAYOFF, DEFAULT_REWARD_PAYOFF, ↩
                DEFAULT_PUNISHMENT_PAYOFF, DEFAULT_SUCKER_PAYOFF)
81          {
82              // No code
83          }
84          #endregion
85
86          #region methods
87          /// <summary>
88          /// Returns player1's payoff of a match.
89          /// </summary>
90          /// <param name="playerOneChoice"></param>
91          /// <param name="playerTwoChoice"></param>
92          /// <returns></returns>
93          public int returnPayoff(Move playerOneChoice, Move playerTwoChoice)
94          {
95              int payoff = 0;
96
97              switch (playerOneChoice)
98              {
99                  case Move.Cooperate:
100                     // Player 1 cooperates, Player 2 cooperates = Reward payoff
101                     if (playerTwoChoice == Move.Cooperate)
102                     {
103                         payoff = this.Reward;
104                     }
105                     // Player 1 cooperates, Player 2 defects = Sucker's payoff
106                     if (playerTwoChoice == Move.Defect)
107                     {
108                         payoff = this.Sucker;
109                     }
110                     break;
111
112                 case Move.Defect:
113                     // Player 1 defects, Player 2 cooperates = Temptation payoff
114                     if (playerTwoChoice == Move.Cooperate)
115                     {
116                         payoff = this.Temptation;
117                     }
118
119                     // Player 2 defects, Player 2 defects = Punishment payoff
120                     if (playerTwoChoice == Move.Defect)
121                     {
122                         payoff = this.Punishment;
123                     }
124                     break;
125             }
126
127             return payoff;
128         }
129
130         /// <summary>
131         /// Checks the validity of the matrix according to the rules :
132         /// T better than R better than P better than S
133         /// </summary>
134         /// <param name="t">Temptation payoff</param>
135         /// <param name="r">Reward payoff</param>
136         /// <param name="p">Punishment payoff</param>
137         /// <param name="s">Sucker's payoff</param>
138         /// <returns>True if the matrix is valid, false if it is not</returns>
139         public static bool isValid(int t, int r, int p, int s)
140         {
141             bool result = false;
142
143             // First condition of validity
144             if ((t < r) && (r < p) && (p < s))
145             {
146                 if (2 * r < t + s)
147                 {
148                     result = true;
149                 }
150             }
151
```

```
152             return result;
153         }
154 
155         /// Overloaded function that allows isValid to be used on the current object
156         /// <summary>
157         /// Checks the validity of the matrix according to the rules :
158         /// T better than R better than P better than S
159         /// </summary>
160         /// <returns>True if the matrix is valid, false if it is not</returns>
161         public bool isValid()
162         {
163             return PayoffMatrix.isValid(this.Temptation, this.Reward, this.Punishment, ↩
                    this.Sucker);
164         }
165 
166         #endregion
167     }
168 }
```

13.3 Classes d'extensions

13.3.1 ArrayExtensions.cs

```
   /*
    Class           :   ArrayExtensions.cs
    Description     :   Allows the conversion of multidimensional arrays and lists
    Author          :   SEEMULLER Julien
    Date            :   16.05.2017
*/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA
{
    public static class ArrayExtensions
    {
        /// <summary>
        /// Converts the current array ([,]) to a list
        /// </summary>
        /// <param name="inputArray">The 2d array to convert</param>
        /// <returns></returns>
        public static List<Cell> asList(this Cell[,] inputArray)
        {
            List<Cell> output = new List<Cell>();

            for (int i = 0; i < inputArray.GetLength(0); i++)
            {
                for (int j = 0; j < inputArray.GetLength(1); j++)
                {
                    output.Add(inputArray[i, j]);
                }
            }

            return output;
        }

        /// <summary>
        /// Converts a list to a 2D array
        /// </summary>
        /// <param name="inputList">The list to convert</param>
        /// <param name="nbLines">The number of lines of the outputted 2d array</param>
        /// <param name="nbCols">The number of columns of the outputted 2d array</param>
        /// <returns></returns>
        public static Cell[,] asArrayOfArray(this List<Cell> inputList, int nbLines, int nbCols)
        {
            Cell[,] output = new Cell[nbLines, nbCols];

            // Check if the input is valid (check if the number of elements is superior or equal
            // to the number of lines times the number of columns
            if (inputList.Count >= nbLines * nbCols)
            {
                int i = 0;

                for (int y = 0; y < nbLines; y++)
                {
                    for (int x = 0; x < nbCols; x++)
                    {
                        output[y, x] = inputList[i];
                        i++;
                    }
                }
            }
            else
            {
                // Else we throw the user an error
                throw new System.ArgumentException("The number of elements is inferior to the size of the outputted 2d array", "original");
            }

            return output;
        }
    }
}
```

13.3.2 ColorExtensions.cs

```
/*
    Class           :   ColorExtensions.cs
    Description     :   Allows the conversion between Color and string format
    Author          :   Ari ROTH
                        http://stackoverflow.com/questions/2395438/convert-system-drawing-color-to-rgb-
    Date            :   07.03.2010
    Changes         :   Adapted for use with transparency, changed to an extension format (this Color) - SEEMULLER Julien - 28.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA
{
    public static class ColorExtensions
    {
        /// <summary>
        /// Converts a color to Hex format
        /// </summary>
        /// <param name="c">The color to convert</param>
        /// <returns></returns>
        public static string ToHex(this Color c)
        {
            return "#" + c.R.ToString("X2") + c.G.ToString("X2") + c.B.ToString("X2");
        }

        /// <summary>
        /// Converts a color to Hex format with transparency
        /// </summary>
        /// <param name="c">The color to convert</param>
        /// <param name="transparency">The transparency level to apply to the color</param>
        /// <returns></returns>
        public static string ToHex(this Color c, byte transparency)
        {
            return "#" + transparency.ToString("X2") + c.R.ToString("X2") + c.G.ToString("X2") + c.B.ToString("X2");
        }

        /// <summary>
        /// Converts a color to RGB format
        /// </summary>
        /// <param name="c">The color to convert</param>
        /// <returns></returns>
        public static string ToRGB(this Color c)
        {
            return "RGB(" + c.R.ToString() + "," + c.G.ToString() + "," + c.B.ToString() + ")";
        }
    }
}
```

13.3.3 ComboBoxExtensions.cs

```
/*
    Class           :   ComboBoxExtensions.cs
    Description     :   Allows the use of colors inside combo boxes
    Author          :   STEPHENS Rod
                        http://csharphelper.com/blog/2016/03/make-a-combobox-display-colors-or-images-in
    Date            :   29.03.2016
    Changes         :   24.04.2017, Adapted for use with strategies - SEEMULLER Julien
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Text;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
18 using System.Windows.Forms;
19
20 namespace PrisonersDilemmaCA
21 {
22     public static class ComboBoxExtensions
23     {
24         // Margins around owner drawn ComboBoxes.
25         private const int MarginWidth = 6;
26         private const int MarginHeight = 2;
27
28         /// <summary>
29         /// Draw a ComboBox item that is displaying a strategy and its color
30         /// </summary>
31         /// <param name="sender"></param>
32         /// <param name="e"></param>
33         private static void DrawItem(object sender, DrawItemEventArgs e)
34         {
35             if (e.Index < 0) return;
36
37             // Clear the background appropriately.
38             e.DrawBackground();
39
40             // Draw the color sample.
41             int height = e.Bounds.Height - 2 * MarginHeight;
42             Rectangle rectangle = new Rectangle(e.Bounds.X + MarginWidth, e.Bounds.Y + ↩
                   MarginHeight, height, height);
43             ComboBox comboBox = sender as ComboBox;
44             Color color = (comboBox.Items[e.Index] as Strategy).getColor();
45
46             using (SolidBrush brush = new SolidBrush(color))
47             {
48                 e.Graphics.FillRectangle(brush, rectangle);
49             }
50
51             // Outline the sample in black.
52             e.Graphics.DrawRectangle(Pens.Black, rectangle);
53
54             // Draw the color's name to the right.
55             using (Font font = new Font(comboBox.Font.FontFamily, comboBox.Font.Size * ↩
                   0.95f, FontStyle.Regular))
56             {
57                 using (StringFormat sf = new StringFormat())
58                 {
59                     sf.Alignment = StringAlignment.Near;
60                     sf.LineAlignment = StringAlignment.Center;
61                     int x = height + 2 * MarginWidth;
62                     int y = e.Bounds.Y + e.Bounds.Height / 2;
63                     e.Graphics.TextRenderingHint = TextRenderingHint.AntiAliasGridFit;
64                     e.Graphics.DrawString(comboBox.Items[e.Index].ToString(), font, ↩
                           Brushes.Black, x, y, sf);
65                 }
66             }
67
68             // Draw the focus rectangle if appropriate.
69             e.DrawFocusRectangle();
70         }
71
72
73         /// <summary>
74         /// Add a list of strategy to a combobox
75         /// </summary>
76         /// <param name="comboBox">The combobox we apply the function to</param>
77         /// <param name="strats">The strategies to add to the combobox</param>
78         public static void AddStrategies(this ComboBox comboBox, List<Strategy> strats)
79         {
80             // Make the ComboBox owner-drawn.
81             comboBox.DrawMode = DrawMode.OwnerDrawFixed;
82
83             // Add the strategies to the ComboBox's items.
84             foreach (Strategy strat in strats)
85             {
86                 comboBox.Items.Add(strat);
87             }
88
89             // Subscribe to the DrawItem event.
90             comboBox.DrawItem += DrawItem;
91         }
92     }
93 }
```

13.4 Stratégies

13.4.1 Strategy.cs

```
   /*
    Class           :   Strategy.cs
    Description     :   Strategy abstract class, Cf. Strategy design pattern.
                        Used to model other strategies.
    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA
{
    public abstract class Strategy : IComparable
    {
        #region methods
        /// <summary>
        /// Returns the next move of the cell based on its neighbors
        /// </summary>
        /// <param name="cell">The cell using this function</param>
        /// <param name="neighbors">The neighbors of the cell using this function</param>
        /// <returns></returns>
        public abstract Move chooseMove(Cell cell, List<Cell> neighbors);

        /// <summary>
        /// Returns the color associated with the strategy
        /// </summary>
        /// <returns></returns>
        public abstract Color getColor();

        /// <summary>
        /// Returns the name of the strategy if it follows the naming convention loosely
        /// The name of the strategy is taken from the filename
        /// ex : "StratTitForTat.cs" -> "Tit for tat"
        /// </summary>
        /// <returns></returns>
        public override string ToString()
        {
            // Get the name of the current class
            string strategyName = this.GetType().Name;

            // Filter the name (remove "Strat" and use spaces insted of CamelCase)
            strategyName = Regex.Replace(strategyName, "(Strat)", "");
            strategyName = Regex.Replace(strategyName, "([a-z])([A-Z])", "$1 $2");

            return strategyName;
        }

        /// <summary>
        /// Used for sorting, alphanumerical sorting according to the name of the ↩
            strategy
        /// </summary>
        /// <param name="obj"></param>
        /// <returns></returns>
        public int CompareTo(object obj)
        {
            return this.ToString().CompareTo((obj as Strategy).ToString());
        }
        #endregion
    }
}
```

13.4.2 StratAdaptativePavlov.cs

```
   /*
    Class           :   StratPavlov.cs
    Description     :   Identifies an opponents according to his moves and counters them
                        http://www.prisoners-dilemma.com/strategies.html

```

```
6       Author          :   SEEMULLER Julien
7       Date            :   10.04.2017
8   */
9
10  using System;
11  using System.Collections.Generic;
12  using System.Drawing;
13  using System.Linq;
14  using System.Text;
15  using System.Threading.Tasks;
16
17  namespace PrisonersDilemmaCA
18  {
19      public class StratAdaptativePavlov : Strategy
20      {
21          #region fields
22          #region consts
23          private const int DEFAULT_NB_OF_ANALYSING_TURNS = 7;
24          #endregion
25
26          private StratTitForTat _tft;
27          private StratTitForTwoTats _tftt;
28          private StratAlwaysDefect _ad;
29          private Strategy _currentStrategy;
30
31          private int _defectCount;
32          #endregion
33
34          #region properties
35          public StratTitForTat Tft
36          {
37              get
38              {
39                  return _tft;
40              }
41
42              set
43              {
44                  _tft = value;
45              }
46          }
47
48          public StratTitForTwoTats Tftt
49          {
50              get
51              {
52                  return _tftt;
53              }
54
55              set
56              {
57                  _tftt = value;
58              }
59          }
60
61          public StratAlwaysDefect Ad
62          {
63              get
64              {
65                  return _ad;
66              }
67
68              set
69              {
70                  _ad = value;
71              }
72          }
73
74          public Strategy CurrentStrategy
75          {
76              get
77              {
78                  return _currentStrategy;
79              }
80
81              set
82              {
83                  _currentStrategy = value;
84              }
85          }
86
87          public int DefectCount
88          {
89              get
```

```
90              {
91                  return _defectCount;
92              }
93
94              set
95              {
96                  _defectCount = value;
97              }
98          }
99          #endregion
100
101         #region constructors
102         public StratAdaptativePavlov()
103         {
104             this.Tft = new StratTitForTat();
105             this.Tftt = new StratTitForTwoTats();
106             this.Ad = new StratAlwaysDefect();
107             this.CurrentStrategy = this.Tft;
108
109             this.DefectCount = 0;
110         }
111         #endregion
112
113         #region methods
114         public override Move chooseMove(Cell cell, List<Cell> neighbors)
115         {
116             // Count the number of defectors before proceeding
117             foreach (var neighbor in neighbors)
118             {
119                 if (neighbor.History.Count > 0)
120                 {
121                     if (neighbor.History.First() == Move.Defect)
122                     {
123                         this.DefectCount++;
124                     }
125                 }
126             }
127
128             // We analyse other cells while playing tit for tat before we reach the ↩
                    threshold
129             if (cell.History.Count < DEFAULT_NB_OF_ANALYSING_TURNS)
130             {
131                 this.CurrentStrategy = this.Tft;
132             }
133             else
134             {
135                 // Change our move only every x rounds
136                 if (cell.History.Count % DEFAULT_NB_OF_ANALYSING_TURNS == 0)
137                 {
138                     // Find the average defect count over the number of analysing ↩
                            turns turns
139                     this.DefectCount /= neighbors.Count;
140
141                     // Choose a move according to the defect count
142                     if (this.DefectCount > 4)
143                     {
144                         // Opponent always defects, we play always defect
145                         this.CurrentStrategy = this.Ad;
146                     }
147                     else if (this.DefectCount == 3)
148                     {
149                         // Opponent is STFT, we play TFTT
150                         this.CurrentStrategy = this.Tftt;
151                     }
152                     else if (this.DefectCount == 0)
153                     {
154                         // Opponent cooperates, we play TFT
155                         this.CurrentStrategy = this.Tft;
156                     }
157                     else
158                     {
159                         // Classified as random strategy, we always defect
160                         this.CurrentStrategy = this.Ad;
161                     }
162
163                     // When we are done analysing, we reset the counter
164                     this.DefectCount = 0;
165                 }
166             }
167
168             // Return our current choice according to our strategy
169             return this.CurrentStrategy.chooseMove(cell, neighbors);
170         }
171
```

```
172         public override Color getColor()
173         {
174             return Color.FromArgb(165, 214, 167);
175         }
176         #endregion
177     }
178 }
```

13.4.3 StratAlwaysCooperate.cs

```
1    /*
2     Class            :   StratAlwaysCooperate.cs
3     Description      :   Always cooperate strategy
4     Author           :   SEEMULLER Julien
5     Date             :   10.04.2017
6 */
7
8 using System;
9 using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Text;
13 using System.Threading.Tasks;
14
15 namespace PrisonersDilemmaCA
16 {
17     public class StratAlwaysCooperate : Strategy
18     {
19         #region fields
20         #endregion
21
22         #region properties
23         #endregion
24
25         #region constructors
26         #endregion
27
28         #region methods
29         public override Move chooseMove(Cell cell, List<Cell> neighbors)
30         {
31             return Move.Cooperate;
32         }
33
34         public override Color getColor()
35         {
36             return Color.FromArgb(46, 204, 113);
37         }
38         #endregion
39     }
40 }
```

13.4.4 StratAlwaysDefect.cs

```
1    /*
2     Class            :   StratAlwaysDefect.cs
3     Description      :   Always defects strategy
4     Author           :   SEEMULLER Julien
5     Date             :   10.04.2017
6 */
7
8 using System;
9 using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Text;
13 using System.Threading.Tasks;
14
15 namespace PrisonersDilemmaCA
16 {
17     public class StratAlwaysDefect : Strategy
18     {
19         #region fields
20         #endregion
21
22         #region properties
```

```
        #endregion

        #region constructors
        #endregion

        #region methods
        public override Move chooseMove(Cell cell, List<Cell> neighbors)
        {
            return Move.Defect;
        }

        public override Color getColor()
        {
            return Color.FromArgb(192, 57, 43);
        }
        #endregion
    }
}
```

13.4.5 StratBlinker.cs

```
   /*
    Class           :   StratBlinker.cs
    Description     :   Blinker strategy, alternates between "defect" and "cooperate"
    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA
{
    public class StratBlinker : Strategy
    {
        #region fields
        #endregion

        #region properties
        #endregion

        #region constructors
        #endregion

        #region methods
        public override Move chooseMove(Cell cell, List<Cell> neighbors)
        {
            Move result;

            if (cell.History.Count % 2 == 0)
            {
                result = Move.Cooperate;
            }
            else
            {
                result = Move.Defect;
            }


            return result;
        }

        public override Color getColor()
        {
            return Color.FromArgb(155, 89, 182);
        }
        #endregion
    }
}
```

13.4.6 StratFortress.cs

```
1   /*
2    Class            :   StratFortress.cs
3    Description      :   Fortress strategy, tries to find neighbors using fortress
4                         and cooperates with them.
5    Author           :   SEEMULLER Julien
6    Date             :   10.04.2017
7   */
8
9   using System;
10  using System.Collections.Generic;
11  using System.Drawing;
12  using System.Linq;
13  using System.Text;
14  using System.Threading.Tasks;
15
16  namespace PrisonersDilemmaCA
17  {
18      public class StratFortress : Strategy
19      {
20          #region fields
21          private bool _hasFoundPartners;
22          #endregion
23
24          #region properties
25          public bool HasFoundPartners
26          {
27              get { return _hasFoundPartners; }
28              set { _hasFoundPartners = value; }
29          }
30          #endregion
31
32          #region constructors
33          public StratFortress()
34          {
35              this.HasFoundPartners = false;
36          }
37          #endregion
38
39          #region methods
40          public override Move chooseMove(Cell cell, List<Cell> neighbors)
41          {
42              Move result = Move.Defect;
43
44              // Strats by playing the sequence "defect, defect, cooperate"
45              switch (cell.History.Count)
46              {
47                  case 0:
48                      result = Move.Defect;
49                      break;
50                  case 1:
51                      result = Move.Defect;
52                      break;
53                  case 2:
54                      result = Move.Defect;
55                      break;
56                  case 3:
57                      result = Move.Cooperate;
58                      break;
59                  // On the fourth and following turns, we look at our neighbors and see ↩
                        if there are
60                  // other "Fortress" players
61                  default:
62
63                      foreach (Cell neighbor in neighbors)
64                      {
65                          if (neighbor.History.Count >= 3)
66                          {
67                              if (neighbor.History.ElementAt(0) == Move.Cooperate)
68                              {
69                                  if (neighbor.History.ElementAt(1) == Move.Defect)
70                                  {
71                                      if (neighbor.History.ElementAt(2) == Move.Defect)
72                                      {
73                                          this.HasFoundPartners = true;
74                                      }
75                                  }
76                              }
77                          }
78                      }
79
80                      // If we have found other fortress players, we cooperate, else we ↩
                            always defect
81                      if (HasFoundPartners)
```

```
82                      {
83                          result = Move.Cooperate;
84                      }
85                      else
86                      {
87                          result = Move.Defect;
88                      }
89
90                      break;
91              }
92
93
94              return result;
95          }
96
97          public override Color getColor()
98          {
99              return Color.FromArgb(230, 126, 34);
100         }
101         #endregion
102     }
103 }
```

13.4.7 StratFortress.cs

```
1    /*
2     Class             :    StratFortress.cs
3     Description       :    Fortress strategy, tries to find neighbors using fortress
4                            and cooperates with them.
5     Author            :    SEEMULLER Julien
6     Date              :    10.04.2017
7 */
8
9  using System;
10 using System.Collections.Generic;
11 using System.Drawing;
12 using System.Linq;
13 using System.Text;
14 using System.Threading.Tasks;
15
16 namespace PrisonersDilemmaCA
17 {
18     public class StratFortress : Strategy
19     {
20         #region fields
21         private bool _hasFoundPartners;
22         #endregion
23
24         #region properties
25         public bool HasFoundPartners
26         {
27             get { return _hasFoundPartners; }
28             set { _hasFoundPartners = value; }
29         }
30         #endregion
31
32         #region constructors
33         public StratFortress()
34         {
35             this.HasFoundPartners = false;
36         }
37         #endregion
38
39         #region methods
40         public override Move chooseMove(Cell cell, List<Cell> neighbors)
41         {
42             Move result = Move.Defect;
43
44             // Strats by playing the sequence "defect, defect, cooperate"
45             switch (cell.History.Count)
46             {
47                 case 0:
48                     result = Move.Defect;
49                     break;
50                 case 1:
51                     result = Move.Defect;
52                     break;
53                 case 2:
54                     result = Move.Defect;
55                     break;
```

```
56                  case 3:
57                      result = Move.Cooperate;
58                      break;
59                  // On the fourth and following turns, we look at our neighbors and see ↩
                        if there are
60                  // other "Fortress" players
61                  default:
62
63                      foreach (Cell neighbor in neighbors)
64                      {
65                          if (neighbor.History.Count >= 3)
66                          {
67                              if (neighbor.History.ElementAt(0) == Move.Cooperate)
68                              {
69                                  if (neighbor.History.ElementAt(1) == Move.Defect)
70                                  {
71                                      if (neighbor.History.ElementAt(2) == Move.Defect)
72                                      {
73                                          this.HasFoundPartners = true;
74                                      }
75                                  }
76                              }
77                          }
78
79                          // If we have found other fortress players, we cooperate, else we ↩
                                always defect
80                          if (HasFoundPartners)
81                          {
82                              result = Move.Cooperate;
83                          }
84                          else
85                          {
86                              result = Move.Defect;
87                          }
88
89                          break;
90                  }
91
92
93                  return result;
94              }
95
96              public override Color getColor()
97              {
98                  return Color.FromArgb(230, 126, 34);
99              }
100             #endregion
101         }
102     }
```

13.4.8 StratGrimTrigger.cs

```
1     /*
2      Class            :   StratGrimTrigger.cs
3      Description      :   Grim trigger strategy, cooperates until some neighbor
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6  */
7
8  using System;
9  using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Text;
13 using System.Threading.Tasks;
14
15 namespace PrisonersDilemmaCA
16 {
17     public class StratGrimTrigger : Strategy
18     {
19         #region fields
20         private bool _wasBetrayed;
21         #endregion
22
23         #region properties
24         public bool WasBetrayed
25         {
26             get { return _wasBetrayed; }
27             set { _wasBetrayed = value; }
```

```
28          }
29          #endregion
30
31          #region constructors
32          public StratGrimTrigger()
33          {
34              this.WasBetrayed = false;
35          }
36          #endregion
37
38          #region methods
39          public override Move chooseMove(Cell cell, List<Cell> neighbors)
40          {
41              // Starts by cooperating
42              Move result = Move.Cooperate;
43
44              // Check if we were betrayed in the past
45              if (WasBetrayed)
46              {
47                  result = Move.Defect;
48              }
49              else
50              {
51                  // If we didn't get betrayed yet, we look at our neighbors
52                  if (cell.History.Count > 1)
53                  {
54                      // Look if we got betrayed by a neighbor after our first move
55                      foreach (Cell neighbor in neighbors)
56                      {
57                          if (neighbor.History.First() == Move.Defect)
58                          {
59                              // If we are betrayed, we switch to a "Always Defect" ↩
                                    strategy
60                              this.WasBetrayed = true;
61                              result = Move.Defect;
62                              break;
63                          }
64                      }
65                  }
66              }
67
68              return result;
69          }
70
71          public override Color getColor()
72          {
73              return Color.FromArgb(52, 73, 94);
74          }
75          #endregion
76      }
77  }
```

13.4.9 StratRandom.cs

```
1     /*
2      Class            :   StratRandom.cs
3      Description      :   Random strategy.
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6  */
7
8  using System;
9  using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Text;
13 using System.Threading.Tasks;
14
15 namespace PrisonersDilemmaCA
16 {
17     public class StratRandom : Strategy
18     {
19         #region fields
20         #endregion
21
22         #region properties
23         #endregion
24
25         #region constructors
26         #endregion
```

```
27
28          #region methods
29          public override Move chooseMove(Cell cell, List<Cell> neighbors)
30          {
31              // Make a new unique random number generator
32              Random rng = new Random(Guid.NewGuid().GetHashCode());
33
34              // Make a list with the possible moves
35              List<Move> availableMoves = new List<Move>();
36              availableMoves.Add(Move.Cooperate);
37              availableMoves.Add(Move.Defect);
38
39              // Return a random element in the list
40              return availableMoves[rng.Next(availableMoves.Count)];
41          }
42
43          public override Color getColor()
44          {
45              return Color.FromArgb(41, 128, 185);
46          }
47          #endregion
48      }
49  }
```

13.4.10 StratSuspiciousTitForTat.cs

```
1     /*
2      Class            :    StratTitForTat.cs
3      Description      :    Same as Tit-for-tat strategy, but defects first
4                            http://www.investopedia.com/terms/t/tit-for-tat.asp
5
6      Author           :    SEEMULLER Julien
7      Date             :    10.04.2017
8  */
9
10 using System;
11 using System.Collections.Generic;
12 using System.Drawing;
13 using System.Linq;
14 using System.Text;
15 using System.Threading.Tasks;
16
17 namespace PrisonersDilemmaCA
18 {
19     public class StratSuspiciousTitForTat : Strategy
20     {
21         #region fields
22         #endregion
23
24         #region properties
25         #endregion
26
27         #region constructors
28         #endregion
29
30         #region methods
31         public override Move chooseMove(Cell cell, List<Cell> neighbors)
32         {
33             // Cooperates on first move, then copies his best openent
34             Move result = Move.Defect;
35
36
37             // If this wasn't our first round, we look at our neighbors
38             if (cell.History.Count > 2)
39             {
40                 // We initialise our variables with the first neighbor in the list
41                 result = neighbors[0].History.First();
42                 int min = neighbors[0].Score;
43
44                 foreach (Cell neighbor in neighbors)
45                 {
46                     if (min > neighbor.Score)
47                     {
48                         min = neighbor.Score;
49                         result = neighbor.History.First();
50                     }
51                 }
52             }
53
54             return result;
```

```
55          }
56
57          public override Color getColor()
58          {
59              return Color.FromArgb(239, 154, 154);
60          }
61          #endregion
62      }
63  }
```

13.4.11 StratTitForTat.cs

```
1     /*
2      Class            :    StratTitForTat.cs
3      Description      :    Tit-for-tat strategy
4                             http://www.investopedia.com/terms/t/tit-for-tat.asp
5
6      Author           :    SEEMULLER Julien
7      Date             :    10.04.2017
8  */
9
10 using System;
11 using System.Collections.Generic;
12 using System.Drawing;
13 using System.Linq;
14 using System.Text;
15 using System.Threading.Tasks;
16
17 namespace PrisonersDilemmaCA
18 {
19     public class StratTitForTat : Strategy
20     {
21         #region fields
22         #endregion
23
24         #region properties
25         #endregion
26
27         #region constructors
28         #endregion
29
30         #region methods
31         public override Move chooseMove(Cell cell, List<Cell> neighbors)
32         {
33             // Cooperates on first move, then copies his best openent
34             Move result = Move.Cooperate;
35
36
37             // If this wasn't our first round, we look at our neighbors
38             if (cell.History.Count > 1)
39             {
40                 // We initialise our variables with the first neighbor in the list
41                 result = neighbors[0].History.First();
42                 int min = neighbors[0].Score;
43
44                 foreach (Cell neighbor in neighbors)
45                 {
46                     if (min > neighbor.Score)
47                     {
48                         min = neighbor.Score;
49                         result = neighbor.History.First();
50                     }
51                 }
52             }
53
54             return result;
55         }
56
57         public override Color getColor()
58         {
59             return Color.FromArgb(200, 200, 200);
60         }
61         #endregion
62     }
63 }
```

13.4.12 StratTitForTwoTats.cs

```
   /*
    Class           :   StratTitForTwoTat.cs
    Description     :   Tit-for-two-tats strategy, copies a neighbors if
                        he plays the same move twice in a row.

    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA
{
    public class StratTitForTwoTats : Strategy
    {
        #region fields
        #endregion

        #region properties
        #endregion

        #region constructors
        #endregion

        #region methods
        public override Move chooseMove(Cell cell, List<Cell> neighbors)
        {
            Move result;
            bool hasToDefect = false;

            // If this wasn't our first round, we look at our neighbors, else we cooperate
            if (cell.History.Count > 1)
            {
                // If one of our neighbors defects twice in a row, we
                foreach (Cell neighbor in neighbors)
                {
                    // Check if our neighbor has played at least 2 turns before proceeding
                    if (neighbor.History.Count >= 2)
                    {
                        if (neighbor.History.ElementAt(0) == Move.Defect && neighbor.History.ElementAt(1) == Move.Defect)
                        {
                            hasToDefect = true;
                            break;
                        }
                    }
                }
            }

            // Send back the correct result
            if (hasToDefect)
            {
                result = Move.Defect;
            }
            else
            {
                result = Move.Cooperate;
            }


            return result;
        }

        public override Color getColor()
        {
            return Color.FromArgb(100, 100, 100);
        }
        #endregion
    }
}
```

13.5 Enums

13.5.1 Enums.cs

```
1   /*
2    Class            :   Enum.cs
3    Description      :   Replaces values with a more verbose alternative
4    Author           :   SEEMULLER Julien
5    Date             :   10.04.2017
6 */
7 using System;
8 using System.Collections.Generic;
9 using System.Linq;
10 using System.Text;
11 using System.Threading.Tasks;
12
13 namespace PrisonersDilemmaCA
14 {
15     // The different moves a cell can play
16     public enum Move { Cooperate, Defect }
17
18     // Defines if the color of the cell is from its actions or strategy
19     public enum ColorMode { Strategy, Playing }
20
21     // Defines if we wrap around the board to find neighbors (like a torus)
22     public enum WrapMode { Default, Torus }
23 }
```

13.6 Tests

13.6.1 CellTests.cs

```
  using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PrisonersDilemmaCA;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Drawing;
using System.Xml;

namespace PrisonersDilemmaCA.Tests
{
    [TestClass()]
    public class CellTests
    {
        [TestMethod()]
        public void DesignatedConstructorTest()
        {
            int x = 0; int y = 0;
            Cell myCell = new Cell(x, y, new StratTitForTat(), new PayoffMatrix());

            Assert.AreEqual(x, myCell.X);
            Assert.AreEqual(y, myCell.Y);
            Assert.AreEqual(new StratTitForTat().GetType(), myCell.Strategy.GetType());
        }

        [TestMethod()]
        public void ConvenianceConstructorTest()
        {
            int x = 0; int y = 0;
            Cell myCell = new Cell(x, y, new PayoffMatrix());

            Assert.AreEqual(x, myCell.X);
            Assert.AreEqual(y, myCell.Y);
        }

        [TestMethod()]
        public void onClickTest()
        {
            Grid myGrid = new Grid(100, 100, 1, 2, new PayoffMatrix());
            StratRandom rndStrat = new StratRandom();

            // Click outside the cell
            myGrid.onClick(60, 60, rndStrat);

            // Compare the strategies names (SHOULD BE NOT EQUAL)
            Assert.AreNotEqual(rndStrat.ToString(), myGrid.Cells[0, ↩
                0].Strategy.ToString());

            // Click inside the cell
            myGrid.onClick(20, 20, rndStrat);

            // Compare the strategies names (SHOULD BE EQUAL)
            Assert.AreEqual(rndStrat.ToString(), myGrid.Cells[0, 0].Strategy.ToString());
        }

        [TestMethod()]
        public void ImplicitConversionTest()
        {
                                                                //  x    y      x    y
            Cell myCell = new Cell(1, 1, new PayoffMatrix());   // [10, 10] to [20, 20]
            myCell.Width = 10;
            myCell.Height = 10;
            Rectangle expected = new Rectangle(10, 10, 10, 10); // [10, 10] to [20, 20]
            Rectangle actual = myCell;

            Assert.AreEqual(expected.X, actual.X);
            Assert.AreEqual(expected.Y, actual.Y);
            Assert.AreEqual(expected.Width, actual.Width);
            Assert.AreEqual(expected.Height, actual.Height);
        }
    }
}
```

13.6.2 ColorExtensionsTests.cs

```
  using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PrisonersDilemmaCA;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Drawing;
namespace PrisonersDilemmaCA.Tests
{
    [TestClass()]
    public class ColorExtensionsTests
    {
        [TestMethod()]
        public void ToHexTest()
        {
            Color actual = Color.FromArgb(255,255,255);
            string expected = "#FFFFFF";
            Assert.AreEqual(expected, actual.ToHex());

            actual = Color.FromArgb(0, 0, 0);
            expected = "#000000";
            Assert.AreEqual(expected, actual.ToHex());
        }

        [TestMethod()]
        public void ToHexTransparentTest()
        {
            Color actual = Color.FromArgb(255, 255, 255, 255);
            string expected = "#FFFFFFFF";
            Assert.AreEqual(expected, actual.ToHex(255));

            actual = Color.FromArgb(255, 0, 0, 0);
            expected = "#FF000000";
            Assert.AreEqual(expected, actual.ToHex(255));
        }

        [TestMethod()]
        public void ToRGBTest()
        {
            Color actual = Color.FromArgb(255, 255, 255);
            string expected = "RGB(255,255,255)";
            Assert.AreEqual(expected, actual.ToRGB());

            actual = Color.FromArgb(0, 0, 0);
            expected = "RGB(0,0,0)";
            Assert.AreEqual(expected, actual.ToRGB());
        }
    }
}
```

13.6.3 ComboBoxExtensionsTests.cs

```
  using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PrisonersDilemmaCA;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Windows.Forms;
namespace PrisonersDilemmaCA.Tests
{
    [TestClass()]
    public class ComboBoxExtensionsTests
    {
        [TestMethod()]
        public void AddStrategiesTest()
        {
            List<Strategy> availableStrategies = new List<Strategy>();
            availableStrategies.Add(new StratTitForTat());
            availableStrategies.Add(new StratAlwaysDefect());
            availableStrategies.Add(new StratAlwaysCooperate());

            ComboBox comboBox = new ComboBox();
            comboBox.AddStrategies(availableStrategies);

```

```
25              int i = 0;
26              foreach (Strategy strat in availableStrategies)
27              {
28                  Assert.AreEqual(strat.ToString(), comboBox.Items[i].ToString());
29                  i++;
30              }
31          }
32      }
33  }
```

13.6.4 GridTests.cs

```
1   using   System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using PrisonersDilemmaCA;
7  using Microsoft.VisualStudio.TestTools.UnitTesting;
8  using System.Drawing;
9  namespace PrisonersDilemmaCA.Tests
10 {
11     [TestClass()]
12     public class GridTests
13     {
14         [TestMethod()]
15         public void DesignatedConstructorTest()
16         {
17             Grid myGrid = new Grid(50, 100, 10, 20, new PayoffMatrix());
18
19             // Check if we have the correct number of cells
20             int expectedNbCells = 10 * 20;
21             Assert.AreEqual(expectedNbCells, myGrid.Cells.Length);
22
23             // Check if the values sent are correct
24             Assert.AreEqual(50, myGrid.Width);
25             Assert.AreEqual(100, myGrid.Height);
26             Assert.AreEqual(10, myGrid.NbLines);
27             Assert.AreEqual(20, myGrid.NbCols);
28         }
29
30         [TestMethod()]
31         public void getPointClampedInGridTest()
32         {
33             Grid myGrid = new Grid(100, 100, 10, 10, new PayoffMatrix());
34
35             // Values to send through the function
36             int sentX1 = 12;
37             int sentY1 = 10;
38             int sentX2 = -2;
39             int sentY2 = -1;
40
41             // Get our positions
42             Point actual1 = myGrid.getPointClampedInGrid(sentX1, sentY1);
43             Point actual2 = myGrid.getPointClampedInGrid(sentX2, sentY2);
44             Point expected1 = new Point(2, 0);
45             Point expected2 = new Point(8, 9);
46
47             // Check for equality
48             Assert.AreEqual(expected1.X, actual1.X);
49             Assert.AreEqual(expected1.Y, actual1.Y);
50
51             Assert.AreEqual(expected2.X, actual2.X);
52             Assert.AreEqual(expected2.Y, actual2.Y);
53         }
54
55         [TestMethod()]
56         public void getCellTest()
57         {
58             Grid myGrid = new Grid(100, 100, 10, 10, new PayoffMatrix());
59             Cell actual;
60
61             // Values to send through the function
62             int x1 = 10;
63             int y1 = 10;
64             int x2 = 11;
65             int y2 = 7;
66             int x3 = -1;
67             int y3 = -1;
68
```

```
69              int expectedX1 = 0;
70              int expectedY1 = 0;
71
72              int expectedX2 = 1;
73              int expectedY2 = 7;
74
75              int expectedX3 = 9;
76              int expectedY3 = 9;
77
78              // Compare 1
79              actual = myGrid.getCell(x1, y1);
80              Assert.AreEqual(expectedX1, actual.X);
81              Assert.AreEqual(expectedY1, actual.Y);
82
83              // Compare 2
84              actual = myGrid.getCell(x2, y2);
85              Assert.AreEqual(expectedX2, actual.X);
86              Assert.AreEqual(expectedY2, actual.Y);
87
88              // Compare 3
89              actual = myGrid.getCell(x3, y3);
90              Assert.AreEqual(expectedX3, actual.X);
91              Assert.AreEqual(expectedY3, actual.Y);
92          }
93
94          [TestMethod()]
95          public void findCellNeighborsTest()
96          {
97              Grid myGrid = new Grid(100, 100, 20, 20, new PayoffMatrix());
98              List<Cell> actual = myGrid.findCellNeighbors(myGrid.getCell(11, 0));
99
100             // Test the actual number of neighbors
101             // 1) Find the width of the "grid" around our cell (neighbor grid)
102             // 2) Find the area of the grid and substract our own cell (in the center)
103             int diameterOfNeighborGrid = (Grid.NEAREST_NEIGHBOR_RANGE * 2) + 1;
104             int expectedCount = (diameterOfNeighborGrid * diameterOfNeighborGrid) - 1;
105
106             Assert.AreEqual(expectedCount, actual.Count);
107         }
108     }
109 }
```

13.6.5 PayoffMatrixTests.cs

```
1   using   System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using PrisonersDilemmaCA;
7   using Microsoft.VisualStudio.TestTools.UnitTesting;
8   namespace PrisonersDilemmaCA.Tests
9   {
10      [TestClass()]
11      public class PayoffMatrixTests
12      {
13          [TestMethod()]
14          public void DesignatedConstructorTest()
15          {
16              int t = 5; int r = 3; int p = 2; int s = 0;
17              PayoffMatrix myMatrix = new PayoffMatrix(t, r, p, s);
18
19              Assert.AreEqual(t, myMatrix.Temptation);
20              Assert.AreEqual(r, myMatrix.Reward);
21              Assert.AreEqual(p, myMatrix.Punishment);
22              Assert.AreEqual(s, myMatrix.Sucker);
23          }
24
25          [TestMethod()]
26          public void returnPayoffTest()
27          {
28              // Test the 4 cases
29              int t = 5; int r = 3; int p = 2; int s = 0;
30              PayoffMatrix myMatrix = new PayoffMatrix(t, r, p, s);
31
32              // TEMPTATION   (Defect - Cooperate)
33              Assert.AreEqual(t, myMatrix.returnPayoff(Move.Defect, Move.Cooperate));
34
35              // REWARD       (Cooperate - Cooperate)
36              Assert.AreEqual(r, myMatrix.returnPayoff(Move.Cooperate, Move.Cooperate));
```

```
37
38              // PUNISHMENT   (Defect - Defect)
39              Assert.AreEqual(p, myMatrix.returnPayoff(Move.Defect, Move.Defect));
40
41              // SUCKER       (Cooperate - Defect)
42              Assert.AreEqual(s, myMatrix.returnPayoff(Move.Cooperate, Move.Defect));
43          }
44
45          [TestMethod()]
46          public void isValidStaticTest()
47          {
48              int t = 0; int r = 1; int p = 3; int s = 5;
49
50              Assert.AreEqual(true, PayoffMatrix.isValid(t, r, p, s));
51              r = 0;
52              Assert.AreEqual(false, PayoffMatrix.isValid(t, r, p, s));
53              r = 1;
54              Assert.AreEqual(true, PayoffMatrix.isValid(t, r, p, s));
55
56          }
57
58          [TestMethod()]
59          public void isValidConvenianceTest()
60          {
61              int t = 0; int r = 1; int p = 3; int s = 5;
62              PayoffMatrix myMatrix = new PayoffMatrix(t, r, p, s);
63
64              Assert.AreEqual(true, myMatrix.isValid());
65              myMatrix.Reward = 0;
66              Assert.AreEqual(false, myMatrix.isValid());
67              myMatrix.Reward = 1;
68              Assert.AreEqual(true, myMatrix.isValid());
69          }
70      }
71  }
```

13.6.6 StrategyTests.cs

```
1   using   System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using PrisonersDilemmaCA;
7   using Microsoft.VisualStudio.TestTools.UnitTesting;
8   namespace PrisonersDilemmaCA.Tests
9   {
10      [TestClass()]
11      public class StrategyTests
12      {
13          [TestMethod()]
14          public void ToStringTest()
15          {
16              StratAlwaysCooperate actual1 = new StratAlwaysCooperate();
17              string expected1 = "Always Cooperate";
18
19              StratAlwaysDefect actual2 = new StratAlwaysDefect();
20              string expected2 = "Always Defect";
21
22              StratBlinker actual3 = new StratBlinker();
23              string expected3 = "Blinker";
24
25              StratRandom actual4 = new StratRandom();
26              string expected4 = "Random";
27
28              Assert.AreEqual(expected1, actual1.ToString());
29              Assert.AreEqual(expected2, actual2.ToString());
30              Assert.AreEqual(expected3, actual3.ToString());
31              Assert.AreEqual(expected4, actual4.ToString());
32          }
33
34          [TestMethod()]
35          public void CompareToTest()
36          {
37              // The strategies are sorted by name
38              int expected = 0;
39              StratAlwaysCooperate strat1 = new StratAlwaysCooperate();
40              StratAlwaysCooperate strat2 = new StratAlwaysCooperate();
41
42              Assert.AreEqual(expected, strat1.CompareTo(strat2));
```

```
43
44              expected = -1;
45              StratAlwaysDefect strat3 = new StratAlwaysDefect();
46
47              Assert.AreEqual(expected, strat1.CompareTo(strat3));
48
49              expected = 1;
50
51              Assert.AreEqual(expected, strat3.CompareTo(strat1));
52          }
53      }
54  }
```

13.6.7 StratAlwaysCooperateTests.cs

```
1   using   System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using PrisonersDilemmaCA;
7   using Microsoft.VisualStudio.TestTools.UnitTesting;
8   namespace PrisonersDilemmaCA.Tests
9   {
10      [TestClass()]
11      public class StratAlwaysCooperateTests
12      {
13          [TestMethod()]
14          public void chooseMoveTest()
15          {
16              // Initialize
17              Grid myGrid = new Grid(200, 200, 10, 10, new PayoffMatrix());
18              myGrid.onClick(5, 5, new StratAlwaysCooperate());
19              Move expected = Move.Cooperate;
20
21              // Compare the last move with what we expected
22              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
23          }
24
25      }
26  }
```

13.6.8 StratAlwaysDefectTests.cs

```
1   using   System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using PrisonersDilemmaCA;
7   using Microsoft.VisualStudio.TestTools.UnitTesting;
8   namespace PrisonersDilemmaCA.Tests
9   {
10      [TestClass()]
11      public class StratAlwaysDefectTests
12      {
13          [TestMethod()]
14          public void chooseMoveTest()
15          {
16              // Initialize
17              Grid myGrid = new Grid(200, 200, 10, 10, new PayoffMatrix());
18              myGrid.onClick(5, 5, new StratAlwaysDefect());
19              Move expected = Move.Defect;
20
21              // Compare the last move with what we expected
22              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
23          }
24
25
26      }
27  }
```

13.6.9 StratBlinkerTests.cs

```
  using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PrisonersDilemmaCA;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace PrisonersDilemmaCA.Tests
{
    [TestClass()]
    public class StratBlinkerTests
    {
        [TestMethod()]
        public void chooseMoveTest()
        {
            // Initialize
            Grid myGrid = new Grid(200, 200, 10, 10, new PayoffMatrix());
            myGrid.onClick(5, 5, new StratBlinker());
            myGrid.step();

            Move expected = Move.Defect;

            // Compare the last move with what we expected
            //Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Cooperate;

            //Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Defect;

            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
        }

    }
}
```

13.6.10 StratFortressTests.cs

```
  using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PrisonersDilemmaCA;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace PrisonersDilemmaCA.Tests
{
    [TestClass()]
    public class StratFortressTests
    {
        [TestMethod()]
        public void chooseMoveTest()
        {
            // Initialize
            Grid myGrid = new Grid(200, 200, 10, 10, new PayoffMatrix());
            myGrid.onClick(5, 5, new StratFortress());

            Move expected = Move.Defect;

            // Compare the last move with what we expected
            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Defect;

            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Defect;

            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Cooperate;

            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());

        }
    }
```

```
}
```

13.6.11 StratGrimTriggerTests.cs

```
  using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PrisonersDilemmaCA;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace PrisonersDilemmaCA.Tests
{
    [TestClass()]
    public class StratGrimTriggerTests
    {
        [TestMethod()]
        public void chooseMoveTest()
        {
            // Initialize
            Grid myGrid = new Grid(100, 100, 10, 10, new PayoffMatrix());
            myGrid.onClick(5, 5, new StratGrimTrigger());
            Move expected = Move.Cooperate;

            // Compare the last move with what we expected
            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());

            // Add a defector
            myGrid.onClick(15, 5, new StratAlwaysDefect());
            expected = Move.Cooperate;
            myGrid.step();

            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Defect;

            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Defect;

            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
        }
    }
}
```

13.6.12 StratRandomTests.cs

```
  using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PrisonersDilemmaCA;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace PrisonersDilemmaCA.Tests
{
    [TestClass()]
    public class StratRandomTests
    {
        [TestMethod()]
        public void chooseMoveTest()
        {
            // Initialize
            Grid myGrid = new Grid(100, 100, 10, 10, new PayoffMatrix());
            myGrid.onClick(5, 5, new StratRandom());

            // Compare the last move with what we expected
            Move actual = myGrid.Cells[0, 0].History.First();
            Assert.AreEqual(true, ((actual == Move.Defect) || (actual == ↩
                Move.Cooperate)));
        }
    }
}
```

13.6.13 StratSuspiciousTitForTatTests.cs

```
  using   Microsoft.VisualStudio.TestTools.UnitTesting;
using PrisonersDilemmaCA;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA.Tests
{
    [TestClass()]
    public class StratSuspiciousTitForTatTests
    {
        [TestMethod()]
        public void chooseMoveTest()
        {
            // Initialize
            Grid myGrid = new Grid(100, 100, 10, 10, new PayoffMatrix(), ↩
                WrapMode.Default, new StratAlwaysCooperate());
            myGrid.onClick(5, 5, new StratSuspiciousTitForTat());
            Move expected = Move.Defect; // starts by defecting

            // Compare the last move with what we expected
            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());


            // Add a cooperator
            myGrid.onClick(15, 5, new StratAlwaysCooperate());
            expected = Move.Defect;
            myGrid.step();

            // Tit for tat doesn't respond
            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Cooperate;

            // Add a defector
            myGrid.onClick(15, 5, new StratAlwaysDefect());
            expected = Move.Cooperate;
            myGrid.step();

            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Defect;

            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
            myGrid.step();
            expected = Move.Defect;

            // Tit for tats mimics the defector
            Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
        }
    }
}
```

13.6.14 StratTitForTatTests.cs

```
  using   System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using PrisonersDilemmaCA;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace PrisonersDilemmaCA.Tests
{
    [TestClass()]
    public class StratTitForTatTests
    {
        [TestMethod()]
        public void chooseMoveTest()
        {
            // Initialize
            Grid myGrid = new Grid(100, 100, 10, 10, new PayoffMatrix());
            myGrid.onClick(5, 5, new StratTitForTat());
            Move expected = Move.Cooperate;

```

```
21              // Compare the last move with what we expected
22              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
23
24
25              // Add a cooperator
26              myGrid.onClick(15, 5, new StratAlwaysCooperate());
27              expected = Move.Cooperate;
28              myGrid.step();
29
30              // Tit for tat doesn't respond
31              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
32              myGrid.step();
33              expected = Move.Cooperate;
34
35              // Add a defector
36              myGrid.onClick(15, 5, new StratAlwaysDefect());
37              expected = Move.Cooperate;
38              myGrid.step();
39
40              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
41              myGrid.step();
42              expected = Move.Defect;
43
44              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
45              myGrid.step();
46              expected = Move.Defect;
47
48              // Tit for tats mimics the defector
49              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
50          }
51      }
52  }
```

13.6.15 StratTitForTwoTatsTests.cs

```
1   using   System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using PrisonersDilemmaCA;
7   using Microsoft.VisualStudio.TestTools.UnitTesting;
8   namespace PrisonersDilemmaCA.Tests
9   {
10      [TestClass()]
11      public class StratTitForTwoTatsTests
12      {
13          [TestMethod()]
14          public void chooseMoveTest()
15          {
16              // Initialize
17              Grid myGrid = new Grid(100, 100, 10, 10, new PayoffMatrix());
18              myGrid.onClick(5, 5, new StratTitForTwoTats());
19              Move expected = Move.Cooperate;
20
21              // Compare the last move with what we expected
22              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
23
24
25              // Add a cooperator
26              myGrid.onClick(15, 5, new StratAlwaysCooperate());
27              expected = Move.Cooperate;
28              myGrid.step();
29
30              // Tit for two tats doesn't respond
31              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
32              myGrid.step();
33              expected = Move.Cooperate;
34
35              // Add a defector
36              myGrid.onClick(15, 5, new StratAlwaysDefect());
37              expected = Move.Cooperate;
38              myGrid.step();
39
40              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
41              myGrid.step();
42              expected = Move.Defect;
43
44              // Tit for two tats mimics the defector after 2 moves
45              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
```

```
46              myGrid.step();
47              expected = Move.Defect;
48
49              Assert.AreEqual(expected, myGrid.Cells[0, 0].History.First());
50              myGrid.step();
51          }
52      }
53  }
```