CFPT-INFORMATIQUE

Travaux de Diplômes 2017

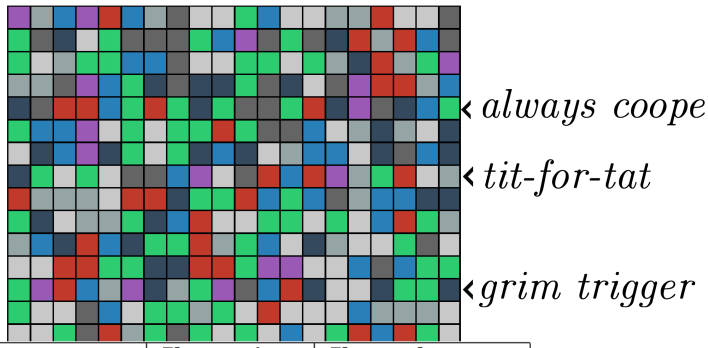
Dilemme du Prisonnier

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

22 mai 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

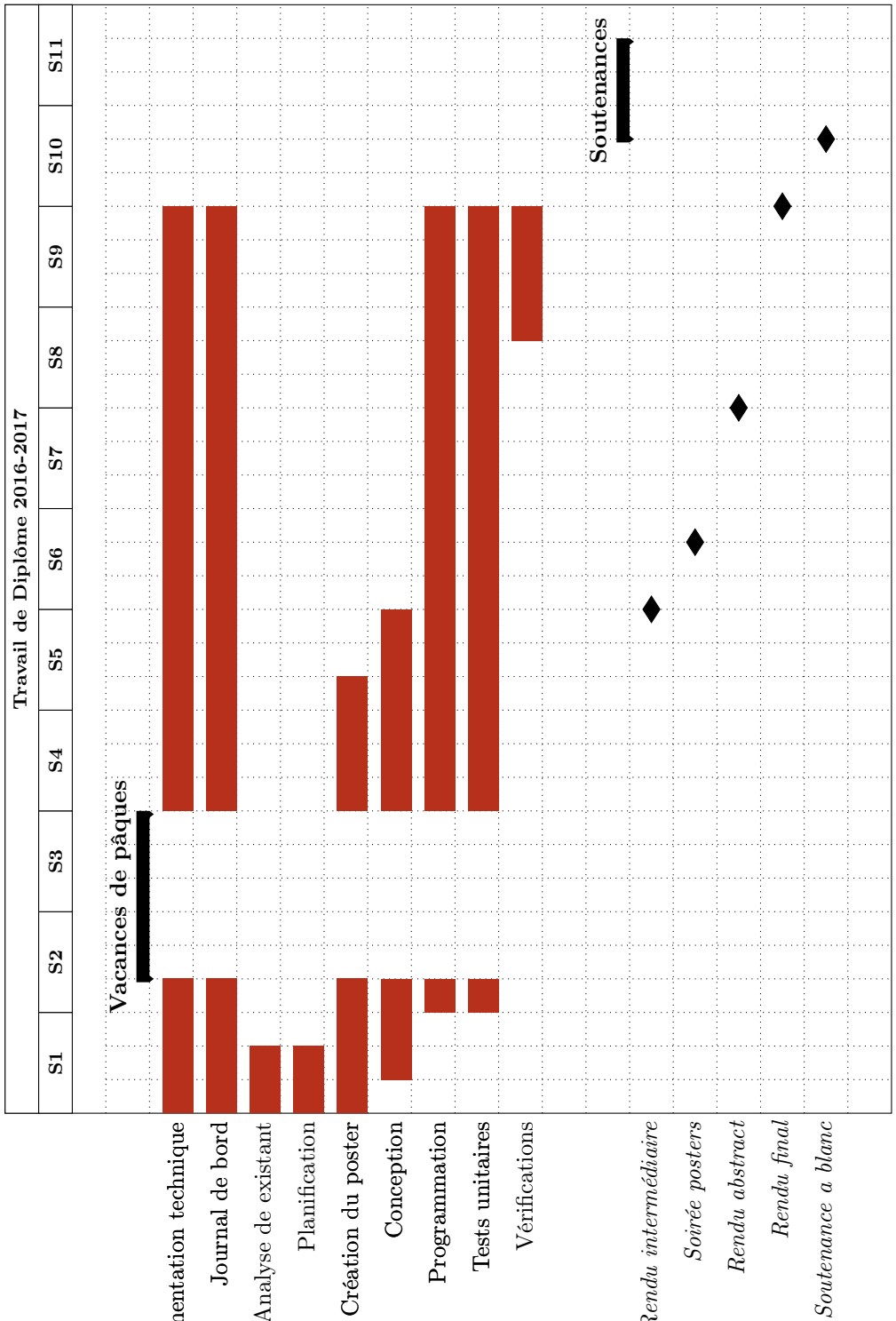


FIGURE 2 – Diagramme de Gantt

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 :

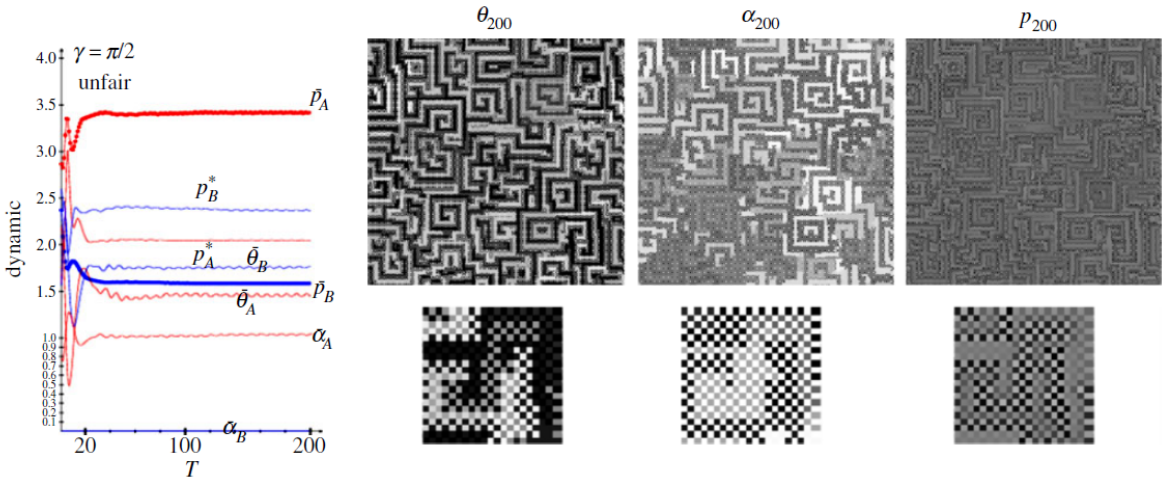


FIGURE 3 – Comparaison de stratégies quantiques (p_A) et classiques (p_B)

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 :

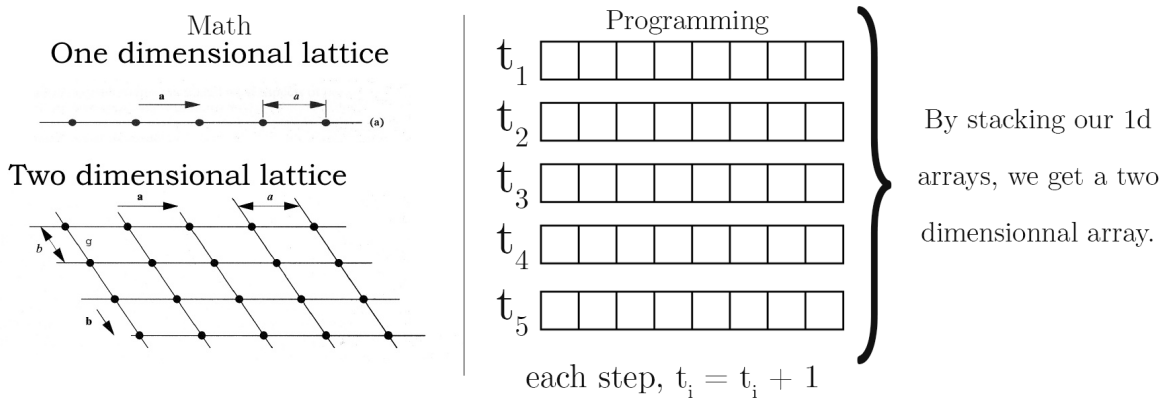


FIGURE 4 – Utiliser la deuxième dimension d'un tableau pour garder un "historique"

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.

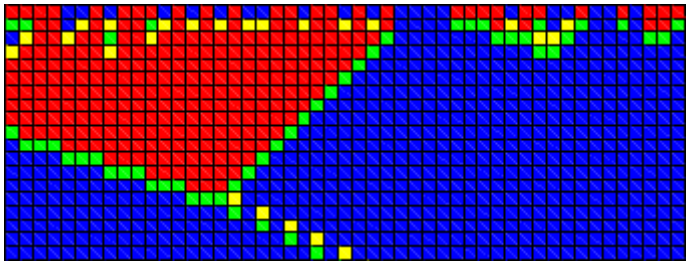


FIGURE 5 – Grappe de cellules "traîtres"

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.

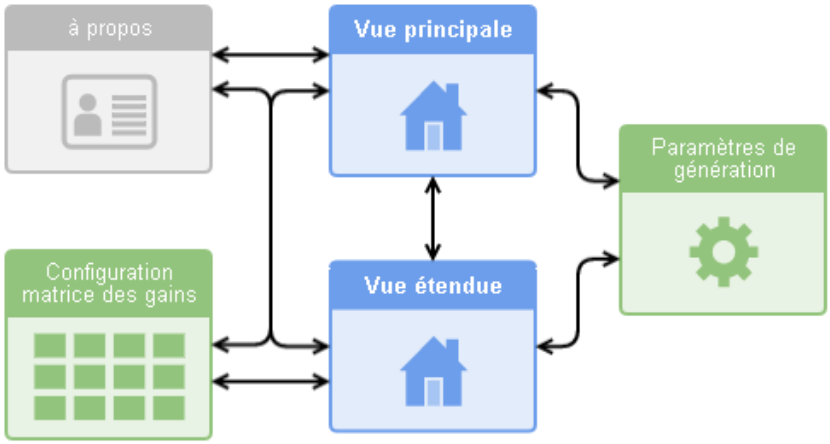


FIGURE 7 – Interactions entre fenêtres de l'application

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.

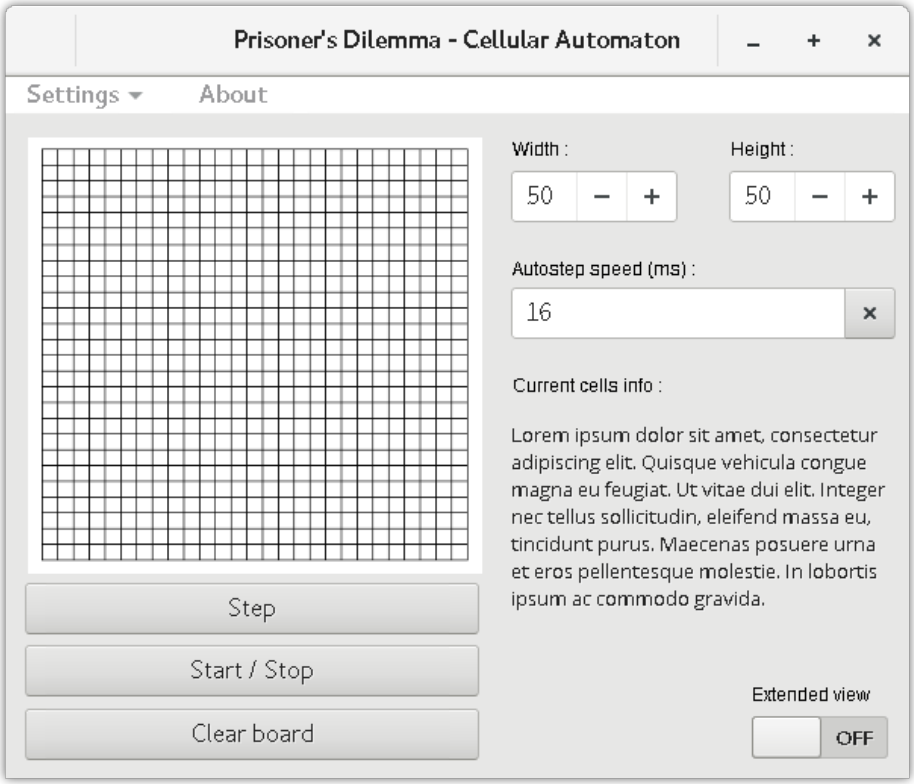


FIGURE 8 – Vue principale de l'application

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*".

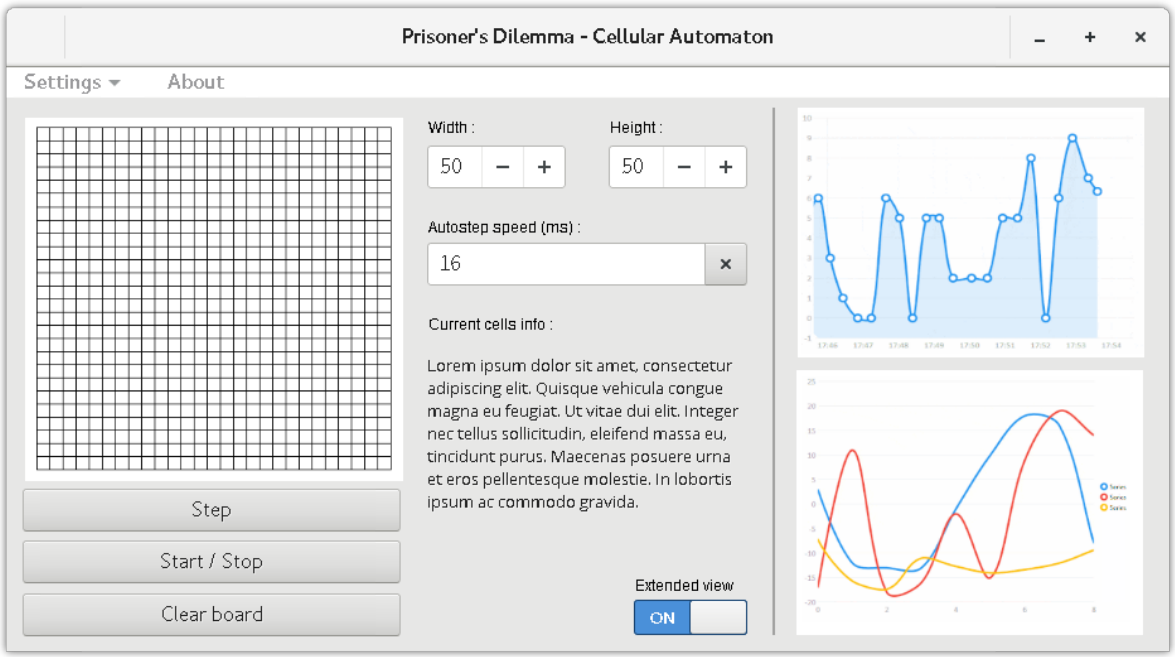


FIGURE 9 – Fenêtre étendue de l'application

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").

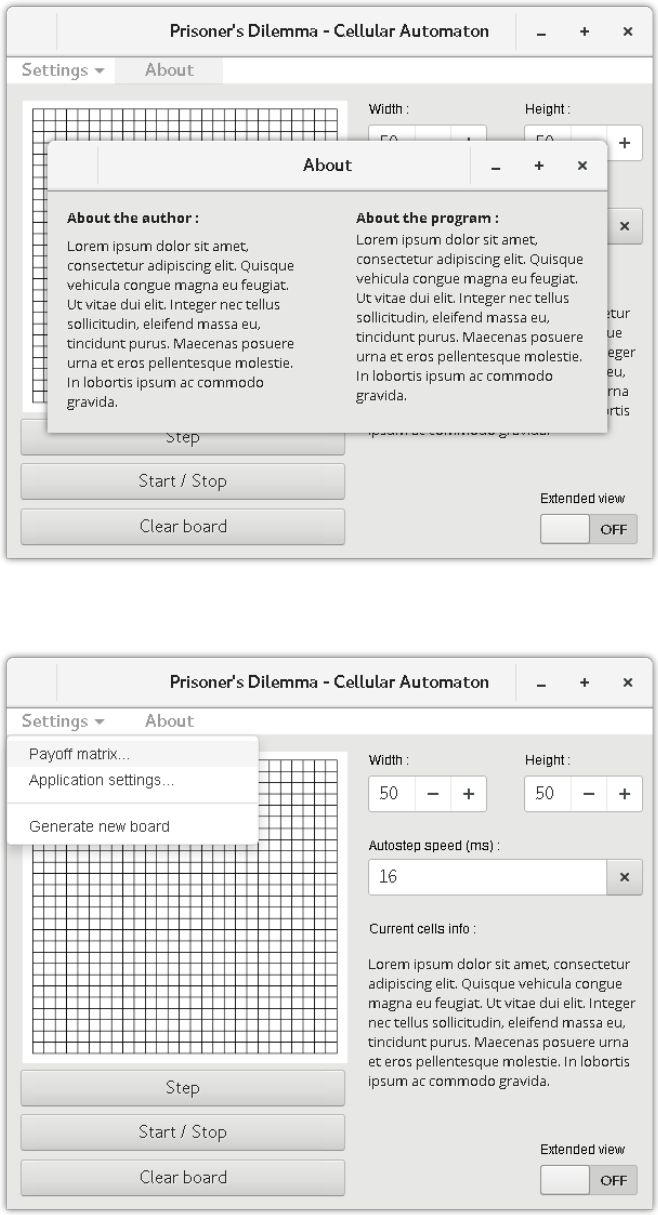


FIGURE 10 – Fenêtre "à propos" et accès aux paramètres de l'application

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.

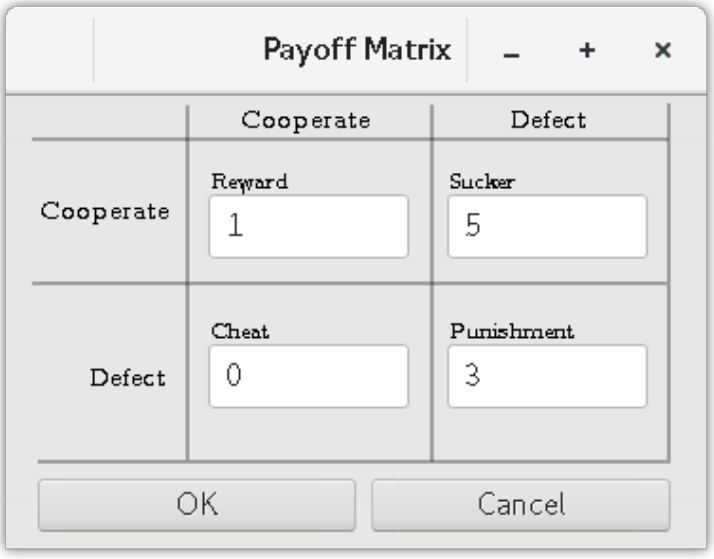


Figure 11 – Configuration de la matrice des gains de l'application

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).

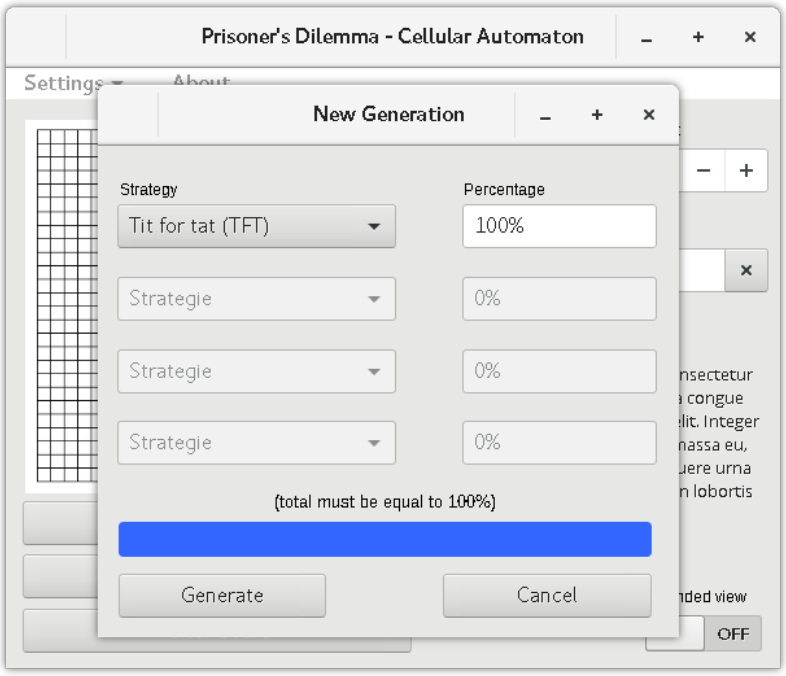


FIGURE 12 – Répartition aléatoire de cellules

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.

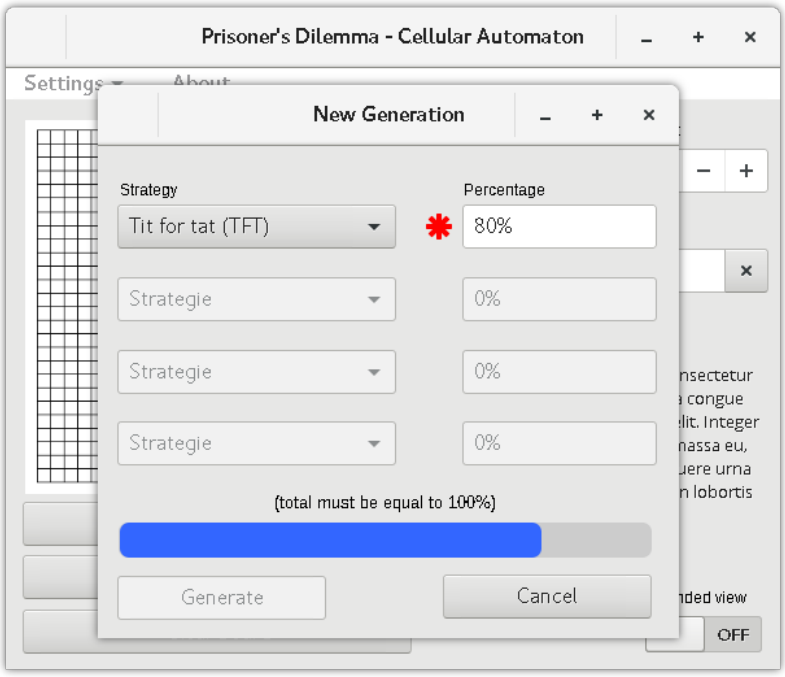


FIGURE 13 – Gestion des erreurs sur la génération aléatoire de cellules

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.

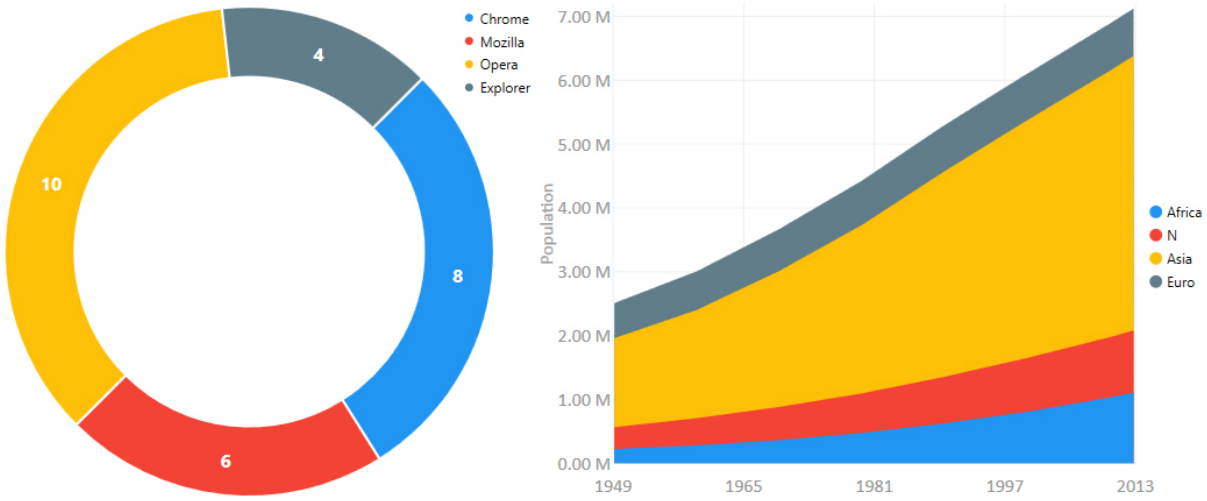


FIGURE 14 – Exemples de graphiques réalisés avec *LiveChars*

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*" :

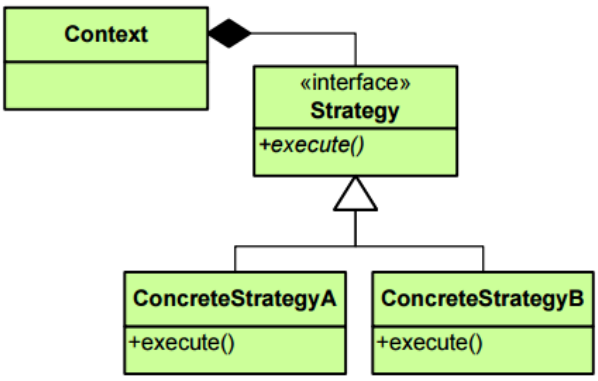


FIGURE 15 – Patron de conception "*Strategy*"

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.

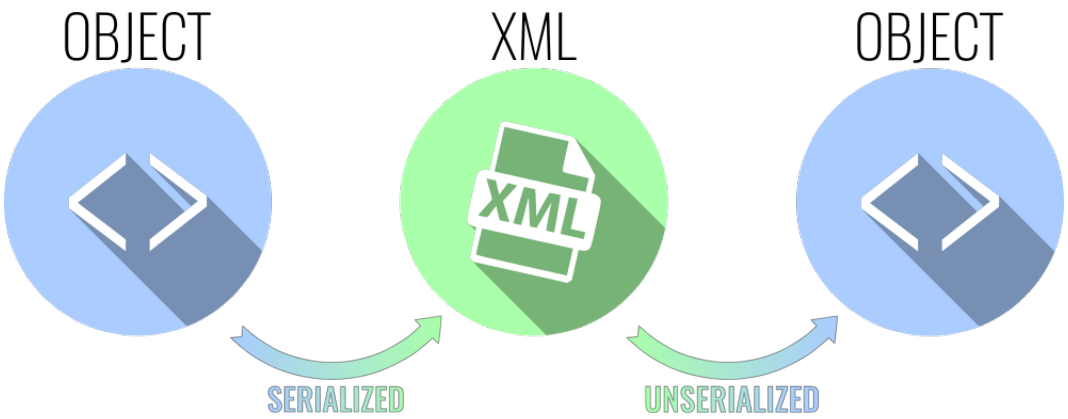


FIGURE 16 – Schéma de sérialisation "*.xml*"

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

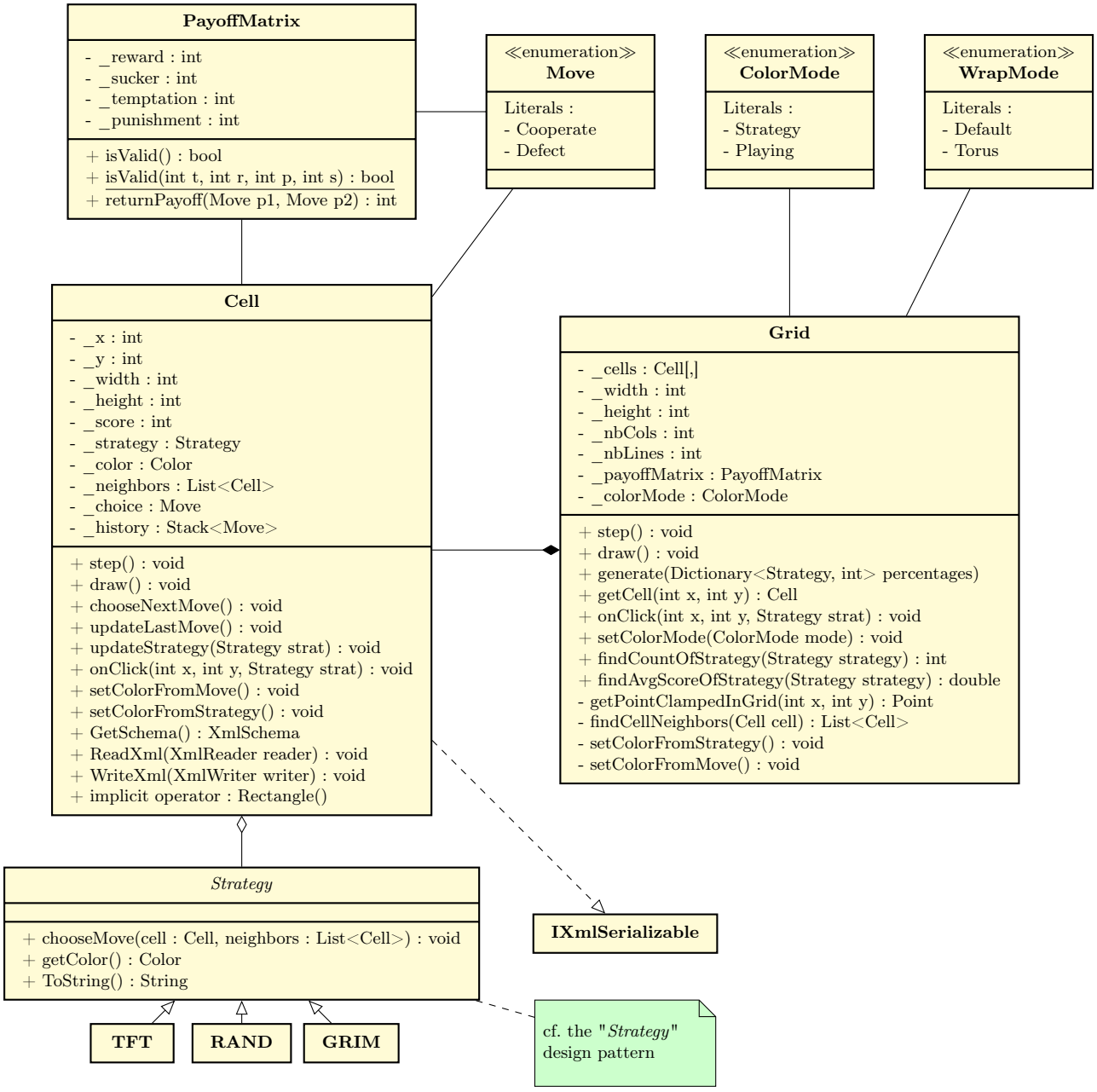


FIGURE 17 – Modèle UML de l'automate cellulaire du dilemme du prisonnier

8.2 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.

8.2.1 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.

8.2.2 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 }
```

8.2.3 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.
`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.

8.2.4 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.

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  }
```

8.2.5 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).

8.2.6 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

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 }
```

8.2.7 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 :

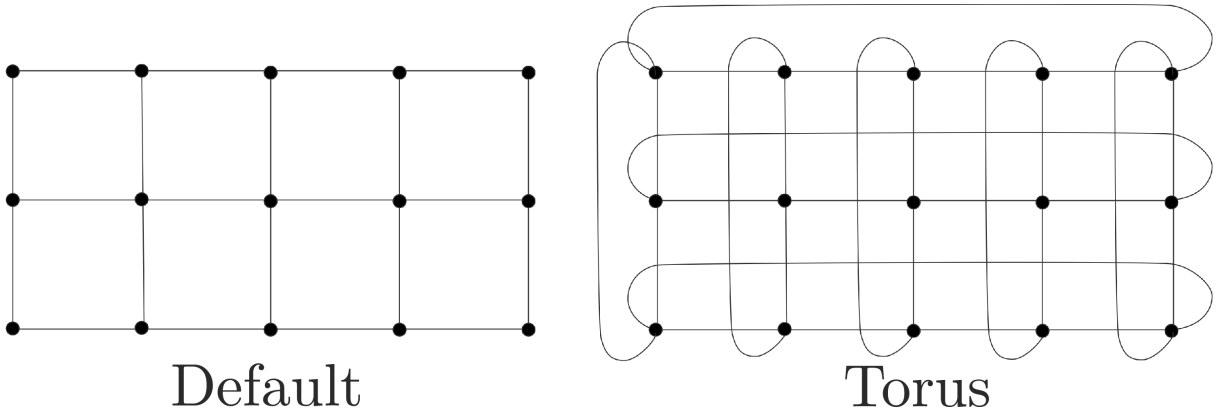


FIGURE 18 – Voisinage des cellules selon le "`WrapMode`"

8.3 Stratégies

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*. Les différentes fonctions prennent en paramètre des tableaux ou des listes d'objets (`objects`) ce qui permet une plus grande flexibilité.

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

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

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
StratTitForTwoTatsTests.chooseMoveTest	00 :00 :00.0011284	Réussite
StratTitForTatTests.chooseMoveTest	00 :00 :00.0012782	Réussite
StratReverseTitForTatTests.chooseMoveTest	00 :00 :00.0016006	Réussite
StratRandomTests.chooseMoveTest	00 :00 :00.0003971	Réussite
StratGrimTriggerTests.chooseMoveTest	00 :00 :00.0012845	Réussite
StrategyTests.ToStringTest	00 :00 :00.0004259	Réussite
StrategyTests.CompareToTest	00 :00 :00.0004124	Réussite
StratBlinkerTests.chooseMoveTest	00 :00 :00.0010023	Réussite
StratAlwaysDefectTests.chooseMoveTest	00 :00 :00.0004947	Réussite
StratAlwaysCooperateTests.chooseMoveTest	00 :00 :00.0043890	Réussite
PayoffMatrixTests.returnPayoffTest	00 :00 :00.0002887	Réussite
PayoffMatrixTests.isValidStaticTest	00 :00 :00.0007546	Réussite
PayoffMatrixTests.isValidConvenianceTest	00 :00 :00.0002266	Réussite
PayoffMatrixTests.DesignatedConstructorTest	00 :00 :00.0003320	Réussite
GridTests.getPointClampedInGridTest	00 :00 :00.0003422	Réussite
GridTests.getCellTest	00 :00 :00.0003659	Réussite
GridTests.findCellNeighborsTest	00 :00 :00.0005940	Réussite
GridTests.DesignatedConstructorTest	00 :00 :00.0019260	Réussite
ComboBoxExtensionsTests.AddStrategiesTest	00 :00 :00.0188054	Réussite
ColorExtensionsTests.ToRGBTest	00 :00 :00.0004100	Réussite
ColorExtensionsTests.ToHexTransparentTest	00 :00 :00.0002800	Réussite
ColorExtensionsTests.ToHexTest	00 :00 :00.0002878	Réussite
CellTests.onClickTest	00 :00 :00.0024636	Réussite
CellTests.ImplicitConversionTest	00 :00 :00.0001942	Réussite
CellTests.DesignatedConstructorTest	00 :00 :00.0092386	Réussite
CellTests.ConvenianceConstructorTest	00 :00 :00.0001753	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]
Interface graphique	90%	Utilisation d'une ".dll" pour le composant "ToggleSwitch"
Graphiques	25%	Utilisation de la bibliothèque "LiveCharts"
Tests unitaires	100%	
Documentation	100%	

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

11 Conclusion et perspectives

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.

Table des figures

Liste des tableaux