

---

# Rapport du projet de programmation et algorithmique

**Semestre 5**

---

**ALOUANI Ismail**

**THIVIN Nathan**

Filière Informatique


## Contents

# 1 Introduction

## 1.1 Contexte

Les automates cellulaires sont au coeur du projet. Ils sont particulièrement connus pour leur utilisation dans le jeu de la vie de John Conway. Il s'agit de cellules évoluant dans un monde en quadrillage en interagissant avec leurs voisins.

Ces objets sont notamment utilisés dans le jeu vidéo Noita évoqué au début du sujet.

 jeu\_de\_la\_vie.png

## 1.2 Le sujet

Le sujet utilise le contexte des automates cellulaires afin de nous faire travailler des éléments d'algorithmique et de programmation impérative vus en cours. Ainsi, il nous a été demandé d'implémenter différents objets autour des automates cellulaires grâce aux structures du C et d'implémenter des structures algorithmique et de réaliser des études de complexité vus en cours d'algorithmique.

## 2 Approche

### 2.1 Outils utilisés et structuration des fichiers

Lors de notre recherche de résolution du problème et de la conception du projet, nous avons utilisé plusieurs outils étudiés lors des cours du 1er semestre.

#### 2.1.1 Git

Pour la gestion de nos dépôts locaux et distants, nous avons opté pour le logiciel de gestion de versions décentralisé GIT, nous permettant une meilleure organisation pour la répartition des tâches.

#### 2.1.2 Emacs

Nous avons utilisé l'éditeur de texte Emacs que nous avons étudié lors du cours d'Environnement de travail pour ses outils facilitant l'écriture du code.

#### 2.1.3 L<sup>A</sup>T<sub>E</sub>X

LaTeX permet une mise en page claire et une facilité d'écriture de documents scientifiques. Nous avons ainsi choisi de l'utiliser pour la rédaction de notre rapport.

#### 2.1.4 Makefile

Un fichier Makefile est un fichier contenant un ensemble de directives utilisées par un outil d'automatisation (make). Ce fichier nous a permis de classer nos fichiers .c et .h de manière à effectuer nos compilations séparées sans nous soucier des dépendances entre nos différents fichiers

### 2.2 Structuration des données

#### 2.2.1 Structure world

Nous avons implémenté notre monde sous la forme d'un tableau d'entiers où l'entier correspond à la couleur de la cellule. Nous avons de même implémenté les fonctions de création et d'affichage d'un monde. Ainsi le struct world adopté tout au long de notre projet aura la forme suivante: struct world unsigned int t[WIDTH\*HEIGHT]; ; i.e. une structure stockant un tableau de nombres positifs qui représentent le code des couleurs qui est la même chose qu'une grille de largeur WIDTH et de longueur HEIGHT donc le tableau aura pour taille HEIGHT\*WIDTH.

### **2.2.2 Structure rule**

Au cours du projet nous avons donné différentes implémentations des règles d'évolution des cellules, nous avons d'abord opté pour une liste chaînée de structures 'rule' avant de nous tourner vers un tableau de structures que nous trouvions plus simple d'utilisation. Nous avons également implémenté les fonctions de manipulation de nos structures. Il n'y a pas de structure commune aux achievements réalisés, ça va être détaillé après.

### **2.2.3 Structure queue**

Le sujet du projet nous demandait d'implémenter une structure de type first in first out, de même que pour les règles nous avons implémenté la file de 2 manières différentes, une première fois avec un tableau et une seconde fois avec une liste chaînée. Nous avons également implémenté les fonctions de manipulation de nos structures. Même remarque apportée que pour le struct rule.

## 3 Résultats

### 3.1 Baseline

#### 3.1.1 Version de base et indéterminisme

A l'aide de nos structures implémentées précédemment nous avons pu définir les règles présentées dans le début du projet. Nous avons également adapté notre structure de règle pour que chaque règle puisse aboutir à plusieurs couleurs différentes.

#### 3.1.2 Le jeu de la vie

##### 1.Parlons de la structure règle considérée ici:

Nous avons adapté notre structure de règle afin d'implémenter le jeu de la vie, il nous paraissait en effet compliqué de définir une à une les règles étant donnée notre implémentation précédente. Comme indiqué partiellement avant, le struct rule du jeu de la vie correspondant à l'achievment 0 aura la forme suivante:

```
struct rule
{
    int couleurs_voisins_utiles[8];
    int nouvelle_couleur[LEN_MAX];
    int own_color;
    struct rule* next;
};
```

i.e. une liste chaînée d'éléments où chaque élément est une structure qui contient 2 informations essentielles: le tableau des voisins; c'est à dire le tableau des couleurs des cases au voisinage d'une case quelconque ; la couleur -1 est prise par convention comme une couleur pas essentielle en terme de voisinage, c'est à dire que le rule\_match prochainement qui devrait tester le voisinage d'une case ne tiendra pas compte de cette case qui est en -1. On part de haut tout a gauche jusqu'en bas tout a droite pour coder les correspondances éléments de tableau couleurs. Dans un espace torique, toute case possède 8 voisins ce qui fera un tableau de 8 éléments (couleurs) pour toute case dans notre grille de jeu(le struct world qu'on va initié aux tests). La règle doit contenir le voisinage, i.e. les couleurs que doit avoir la case pour qu'elle change et forcément la nouvelle couleur à considérer, c'est une implémentation indéterministe donc la nouvelle couleur est dans un ensemble de couleurs possibles donc le champ nouvelle\_couleur doit être un tableau de taille prise LEN\_MAX qui correspond a 100 ici, c'est à dire le maximum des couleurs possibles a prendre en compte avec -1 si couleur à qui on s'intéresse pas en terme d'indices. La structure contient également un champ own\_color qui code la couleur que doit avoir une case. Et finalement, un pointeur vers le prochain élément de même type: struct rule. En général, notre struct rule dans notre jeu de la vie permet de fixer la couleur que doit avoir une case, le

voisinage que doit avoir cette dernière, pour que sa couleur change en une couleur prenant sa valeur dans ce tableau d'entiers nouvelle\_couleur. C'est notre première implémentation de struct rule opérationnel lors des tests automatisés du jeu de la vie de la baseline. Avec cette structure, on peut implémenter les 6 fonctions nécessaires a sa manipulation en respectant les prototypes, cela a été réalisé dans le fichier src/rule.c.

## **2.Parlons de la structure file considérée ici:**

Pour manipuler la file, on a commencé à poser un type de contrat qui est le fichier src/code\_queue.h qui fournit l'implémentation de la file (sans abstraction i.e. pas de type incomplet ici pour la structure de file) qui sera un tableau d'éléments où chaque élément est une structure de type: struct rule\_index qui sera de la forme suivante:

```
struct rule_index
{
    struct rule rules;
    int xy[2];
};
```

i.e. que le type struct rule\_index qui est le type de chaque élément dans le champ 'tableau' de la structure file contient 2 champs principales; celui nécessaire pour savoir la règle et le couple (i,j) (2ième champ) sur laquelle elle va s'appliquer. La structure file sera un tableau d'éléments de ce type, donc aura la forme suivante:

```
struct file
{
    struct rule_index couple_rxy[HEIGHT*WIDTH];
};
```

On stocke au maximum HEIGHT\*WIDTH changements (au maximum toutes les cases changent), donc le tableau sera de taille HEIGHT\*WIDTH et aura pour nom: couple\_rxy.

De même on a répertorié dans le .h toutes les fonctions pour manipuler cette structure à savoir les fonctions de prototypes suivants:

```
void queue_init();
void display_rxy(struct rule_index rxy);
void display_file(struct file f);
int file_vide(struct file f);
void ajout_file(struct file* f, struct rule_index couple);
void delete_file(struct file* f);
void afficher_head_queue(struct file* f);
```

Les tests relatives aux manipulations de la structure rule avec le world, ainsi que la file avec les 2 fichiers précédents (src/rule.c ; src/code\_queue.c) sont suivants les targets test1; test2 dans le Makefile.

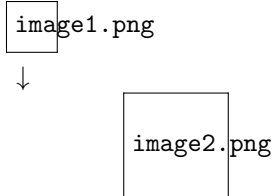
### 3.1.3 Mouvements et incompressibilité

Nous avons du revoir notre structure de règle pour prendre en compte les déplacements des cellules. Nous avons par ailleurs ajouter un système de file afin de gérer les différents mouvements et par la suite les problèmes d'incompressibilité.

## 3.2 Tests

### 3.2.1 Première règle

Nous avons mis en place des tests afin de vérifier le fonctionnement de notre code, ceux-ci sont constitués d'un main appelant nos fonctions afin de créer un monde aléatoire et appliquer à chacune de ses cellules les règles de la version de base avec et sans déterminisme. Ainsi ce dernier correspond au target test3 du Makefile suivant le fichier test\_final\_jeu.c qui est un fichier qui importe le code\_queue.h incluant le rule.h. ça permet de faire un codage pour la règle donnée dans la page du rapport et la tester suivant une fonction void jeu() après avoir fait une initialisation bien maîtrisée pour la structure de la règle et la structure de la file. La procédure pour y arriver est claire, on enfile à chaque fois le couple et la règle à y appliquer; après une application de cet ensemble de règles suivant le champ nouvelle\_couleur en choisissant le premier élément tjrs (indéterminisme), on calcule la longueur des éléments significatifs de la file avec un entier incrémenté au fur et à mesure (int index;), la file est initialisée suivant un tableau correspondant à des (-1,-1);-1, ainsi après l'enfilation après avoir itéré sur tous les (i,j) et incrémenté le index, on applique ces changements avec la transformation  $i * \text{WIDTH} + j$  suivant le champ nouvelle\_couleur[0] parmi les éléments de la file: .couple\_rxy[k] pour k allant de 0 à index à limite pour limite=HEIGHT\*WIDTH; longueur du tableau représentant le world.



### 3.2.2 Jeu de la vie

Nous avons créé un fichier main qui définit un monde de taille demandée et y applique les règles du jeu de la vie. Ainsi dans le fichier project.c (version ancienne de nommage des fichiers), on a importé le code de la file (code\_queue.h et getopt.h pour pouvoir instancier le nombre de frames + le générateur aléatoire(srand(int))) sans oublier la fonction de rule\_matchRqui dresse le tableau de voisinage pour un couple (i,j) donné de la grille puis calcule le tableau de voisinage lui correspondant et après elle calcule suivant ce tableau de voisinage précédemment calculé soit: le nombre de cases blanches si la case est noire(pour qu'elle change potentiellement en BLANC si le nombre



de ces cases est égal à 3) ; (ou qu'elle change en NOIR si le nombre de ces cases est inférieur strictement à 2 (pas de population voisine active) ou supérieur strictement à 3 (sur population de la case par des blancs autour) ). Ainsi un appel à `rule_matchR` renvoie un entier qui est 1 (première règle qui correspond au changement de la case NOIR en BLANC) ou 2 (deuxième règle qui correspond au changement de la case BLANC en NOIR, après pour la fonction `jeu_vie(int)` qui prend en argument un nombre d'itérations, elle fera le même job que le coeur de la fonction présentée précédemment au niveau du fichier `test_final_jeu.c`. Après on applique `HEIGHT*WIDTH`-index fois la fonction `delete_file(struct file*)` pour vider la file et la préparer au prochain tour suivant.(sans oublier de faire un affichage du world au passage avant l'incréméntation du compteur `k` de la boucle `while`)

### **3.2.3 Modélisations des mouvements d'un grain de sable**

Afin de tester nos fonctions d'implémentation du mouvement des cellules nous avons défini des règles modélisant la chute et l'entassement de grains de sable et les avons testé d'un main, le moodèle semble correct malgré quelques incohérences vis-à-vis de la réalité physique.

## 4 Conclusion

```
int a = 0;
while (a < n)
{
    a++;
}
```

