Projet de Programmation Fonctionnelle 2015-2016 : Automates Cellulaires

Table des matières

1 Introduction

(Source : Wikipédia) Un automate cellulaire consiste en une grille de *cellules* contenant chacune un *état* choisi parmi un ensemble fini d'états possibles, qui évolue au cours du temps. L'état d'une cellule au temps $(t+1)$ est fonction de l'état au temps t d'un nombre fini de cellules appelé son *voisinage*. À chaque nouvelle unité de temps, les mêmes règles sont appliquées simultanément à toutes les cellules de la grille, produisant une nouvelle *génération* de cellules dépendant entièrement de la génération précédente [1].

Exemple 1. Un exemple bien connu d'automate cellulaire est le *jeu de la vie* de Conway. Dans cet exemple, le voisinage d'une cellule est l'ensemble contenant la cellule elle même et les

1. Formellement, une génération est une fonction dont le domaine est l'ensemble de cellules et dont le codomaine est l'ensemble des états.

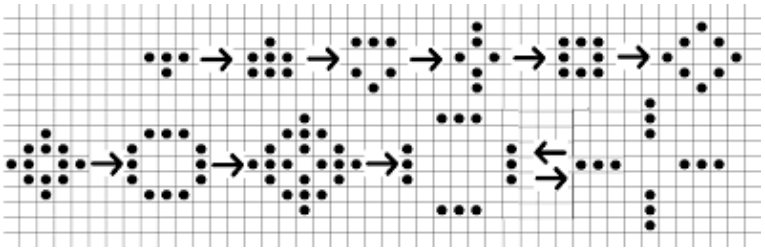


FIGURE 1 – Exemple d'évolution du jeu de la vie

huit cellules qui l'entourent [2]. L'ensemble des états possibles des cellules est $\{A, D\}$ ("*A*live", "*D*ead").

À chaque étape, l'évolution d'une cellule est entièrement déterminée par son état et celui de ses huit voisines de la façon suivante :

— une cellule morte (D) possédant exactement trois voisines vivantes (A) devient vivante (elle naît) ;

— une cellule vivante (A) possédant deux ou trois voisines vivantes (A) le reste, sinon elle meurt.

Malgré la simplicité de cette règle, le jeu de la vie est Turing-complet.

En Figure 1, vous trouvez deux exemples d'évolution du jeu de la vie, les cellules vivantes sont celles qui contiennent un cercle.

Une grille avec un ensemble de cellules vivantes est appelée une *génération*.

Une génération G d'un automate cellulaire est dite *stable* si la génération suivante G' produite à partir de G par application des règles d'évolution de l'automate à toutes les cellules est égale à G. Autrement dit, une génération est stable si elle est un point fixe de la fonction d'évolution de l'automate.

Pour ce projet, vous allez écrire un programme OCaml qui permet de calculer et visualiser l'évolution d'un automate cellulaire, et de calculer et visualiser ses générations stables.

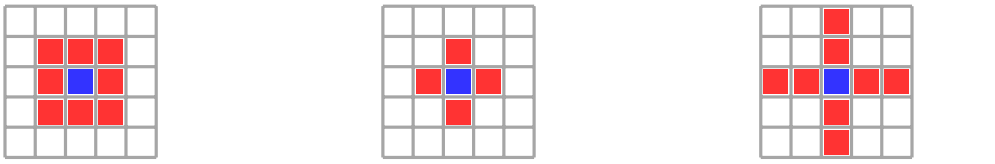
2 Sujet minimal

Nous nous intéresserons tout d'abord au voisinage de rayon 1. Le voisinage de rayon 1 d'une cellule c est composé de c et des 4 cellules adjacentes à c (horizontalement et verticalement), comme en Figure 2b. Ce voisinage particulier s'appelle le *voisinage de Von Neumann de rayon* 1. D'autres voisinages seront utilisés dans des extensions optionnelles de ce sujet (voir Figure 2).

L'ensemble des états est réduit à deux éléments A et D (comme dans le jeu de la vie).

Vous devrez écrire et nous présenter un programme en OCaml permettant de :

2. La grille du jeu est bi-dimensionnelle et théoriquement infinie, donc chaque cellule possède huit voisins. Pour une matrice $n \times m$, les voisins de gauche d'une cellule en première colonne (colonne 0) se trouvent en dernière colonne (colonne $n - 1$), et symétriquement, les voisins de droites d'une cellule en dernière colonne se trouvent en première colonne. Il en est de même pour les lignes.



(a) Moore
(jeu de la vie)

(b) Von Neumann de rayon 1
(sujet minimal)

(c) Von Neumann de rayon 2

FIGURE 2 – Différentes notions de voisinage
(le sujet minimal se réfère seulement au voisinage de Von Neumann de rayon 1).

1. **Initialiser :** lire un fichier texte contenant la spécification d'un automate cellulaire et d'une génération initiale pour cet automate, et sauvegarder ces données dans des valeurs Ocaml de type adéquat.
2. **Afficher :** visualiser en mode texte une génération d'un automate cellulaire.
3. **Simuler :** calculer pour un automate cellulaire et une génération donnés, la génération suivante.
4. **Modéliser en Calcul Propositionnel :** pour un automate *Aut* donné, engendrer une formule du calcul propositionnel telle que chaque affectation des valeurs de vérité satisfaisant la formule corresponde à une génération stable de *Aut*, et vice versa.
5. **Trouver les générations stables :** utiliser le *résolveur SAT* `minisat` pour trouver les affectations satisfaisant la formule du point précédent, et visualiser en mode texte les générations stables correspondantes.

En OCaml, on utilisera les noms de types suivants pour représenter un automate et une génération. À vous de les définir convenablement, ainsi que d'autres types utiles éventuels.

```
type state       (* les etats possibles d'une cellule *)
type generation  (* une generation *)
type rule        (* une regle de l'automate *)
type automaton   (* un automate *)
```

2.1 Initialiser

Le but de cette partie est de définir la fonction [3] :

```
parse : in_channel -> int * automaton * generation
```

qui lit un fichier texte contenant la dimension d'une grille, la spécification d'un automate cellulaire et d'une génération initiale pour cet automate, et renvoie ces données dans des valeurs Ocaml de type adéquat.

Pour simplifier, la grille de l'automate est une matrice carré. Le fichier texte sera dans le format standard suivant.

3. On vous fournit le type des fonctions principales juste comme suggestion. Vous avez la pleine liberté de proposer des variantes, si ceux-là conviennent à votre implémentation.

— Le première ligne du fichier contient l'entier correspondant à la taille de la grille.

— La deuxième ligne contient la chaîne `Regles`. Elle est suivi de lignes spécifiant les règles d'évolution de l'automate. Plus précisément, ces lignes indiquent la liste des voisinages donnant à une cellule l'état de vivant (A). Les voisinages n'apparaissant pas dans cette liste correspondent alors aux cas dans lesquels une cellule évolue ou reste dans l'état mort (D). Le format d'un voisinage sera $EtatNord\ EtatEst\ EtatSud\ EtatOuest\ EtatCellule$, chacun des cinq états valant soit A soit D. Par exemple, la ligne $ADADD$ se lit :

  "Si les voisins Nord et Sud d'une cellule sont dans l'état A, ses voisins Est et Ouest sont dans l'état D, et la cellule est dans l'état D au temps t, alors la cellule sera dans l'état A au temps $(t + 1)$".

— Après les règles, les fichier contient la ligne `GenerationZero`, suivie de la grille correspondant à la génération initiale, ligne par ligne.

Exemple 2. Le fichier :

```
7
Regles
AAAAA
AAAAD
AAADA
AAADD
GenerationZero
DAAAAAA
ADAAAAA
AADAAAA
AAADAAA
AAAADAA
AAAAADA
AAAAAAD
```

spécifie un automate de taille 7, où une cellule nait ou reste vivante si et seulement si ses trois voisins Nord, Est, Sud sont vivants. Dans la génération initiale, toutes les cellules sont vivantes sauf celles de la diagonale principale.

2.2 Afficher

Le but de cette partie est de programmer la fonction :

```
show_generation : generation -> unit
```

La fonction `show_generation` affiche sur l'écran l'état d'une génération donnée en entrée.

2.3 Simuler

Le but de cette partie est de programmer la fonction :

```
next_generation: automaton * generation -> generation
```

La fonction `next_generation` permet de faire évoluer une génération selon les règles de l'automate.

2.4 Modéliser en calcul propositionnel

Le but de cette partie du projet est définir une fonction

```
stables: automaton -> formula
```

qui associe à un automate une formule en forme normale conjonctive [4] caractérisant les générations stables de l'automate donné en entrée.

On peut utiliser le type `formula` vu en TP n°4. L'idée est d'associer une variable propositionelle à chaque case de la grille de l'automate (donc pour un automate sur une grille de dimension 7 on utilisera $7 \times 7 = 49$ variables). Indiquons par $x_{i,j}$ la variable associée à la case de ligne i et colonne j. On encode les deux états `A` et `D` des cellules par les deux valeurs booléennes, respectivement `true` et `false`. Ainsi, une affectation des variables décrit toujours une génération. Par exemple, l'affectation associée à la génération initiale de l'Exemple 2 est :

$$x_{i,j} = \begin{cases} \texttt{false} & \text{si } i{=}j, \\ \texttt{true} & \text{sinon.} \end{cases}$$

Le but est de trouver, étant donné un automate `aut` sur une grille d'une certaine dimension `d`, une formule en forme normale conjonctive $F_{\texttt{aut,d}}$ exprimant qu'une génération est stable (au sens défini plus haut), c.à.d. une affectations satisfait $F_{\texttt{aut,d}}$ *si et seulement si* elle est associée à une génération stable de `aut`.

Remarquez que cette formule $F_{\texttt{aut,d}}$ ne dépend pas de la génération initiale, mais seulement de l'automate et de la dimension de la grille.

Suggestion : on peut se contenter de produire la liste des clauses disjonctives de $F_{\texttt{aut,d}}$. Dans ce cas, la fonction principale est `stables: automaton -> formula list`. La formule $F_{\texttt{aut,d}}$ s'obtenant simplement par un *fold* du connecteur `Et` sur la liste résultat. Les deux choix d'implémentation de la fonction `stables` sont donc acceptables.

2.5 Trouver les générations stables

2.5.1 Brève introduction à *minisat*

Vous allez utiliser le résolveur *minisat* [5] qui permet de déterminer la satisfaisabilité de formules propositionnelles en forme normale conjuctive.

Minisat accepte en entrée des formules représentées au format *DIMACS*, qui est un format standard de description de formules en forme normale conjonctive. Voici un exemple de fichier au format *DIMACS*,

4. On rappelle qu'une formule en forme normale conjonctive est une conjonction de disjonctions des littéraux :

$$(\ell_{1,1} \vee \cdots \vee \ell_{1,k_1}) \wedge \cdots \wedge (\ell_{n,1} \vee \cdots \vee \ell_{n,k_n})$$

où un littéral $\ell_{i,j}$ est une variable ou la négation d'une variable. Les disjonctions $\ell_{i,1} \vee \cdots \vee \ell_{i,k_i}$ sont aussi appelées *clauses* de la formule.

5. http ://www.minisat.se/ ; *minisat* est gratuit et installé sur les machines de l'UFR.

```
p cnf 3 4
3 0
1 2 0
1 -3 0
2 3 -1 0
```

La première ligne indique, après les mots clef `p cnf` (propositional conjunctive normal form) le nombre de variables intervenant dans la formule (ici 3), ainsi que le nombre de clauses disjonctives (ici 2). Chaque ligne suivante représente une clause disjonctive : les variables apparaissant dans la clause sont indiquées par un nombre (ici 1, 2 ou 3), avec un signe moins si elles apparaissent sous forme de négation. Chaque ligne se termine par un 0. Le fichier ci-dessus correspond donc à la formule suivante :

$$x_3 \wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee \neg x_1)$$

La commande

```
minisat entree.dimacs sortie
```

lance l'exécution de *minisat* sur le fichier d'entrée `entree.dimacs` et produit le fichier `sortie`, contenant une affectation des variables qui satisfait la formule donnée en entrée, si elle existe, sinon une déclaration de non satisfaisabilité . Par exemple, si on appelle `minisat` sur le fichier ci-dessus, on aura le résultat suivant :

```
SAT
1 -2 3 0
```

`SAT` signifie que la formule est satisfaisable. La ligne `1 -2 3` signifie que l'affectation $[x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto 1]$ est une solution au problème. En revanche, avec la formule

$$x_3 \wedge \neg x_3$$

`minisat` donnera

```
UNSAT
```

qui signifie que la formule n'est pas satisfaisable.

2.5.2 Générations stables

La fonction principale à programmer pour cette partie du projet est :

```
show_stable: unit -> unit
```

qui crée un fichier contentant la formule de la section 2.4 au format *DIMACS*, lance *minisat* sur ce fichier, récupère et parse le résultat de *minisat*, affiche la génération stable correspondante (ou termine), et finalement, met à jour le contenu de la liste de clauses disjonctives décrite plus haut, pour éliminer la génération qui vient d'être calculée.

Plus précisément, pour cette partie du projet il faudra :

1. Convertir la formule de la section 2.4 (qui doit être en forme normale conjonctive) au format *DIMACS*, et l'écrire dans un fichier `entree.dimacs`.

   On pourra donc imaginer de mettre en ouvre une fonction `create_dimacs` de type `formula -> out_channel -> unit`, ou bien `formula list -> out_channel -> unit`, en fonction du choix effectué à la partie 2.4
2. Lancer *minisat* sur `entree.dimacs`, et récupérer la sortie correspondante. Pour exécuter un commande sur la shell à partir d'un programme d'Ocaml, utilisez la fonction `command` du module `Sys` de la librairie standard de Ocaml.
3. Si le résultat est `UNSAT`, annoncer qu'il n'y a pas (ou plus) de génération stable, sinon lire l'affectation produite par *minisat* et afficher la génération stable correspondante.
4. Si l'utilisateur le demande, reprendre la recherche des générations stables à partir de 2. (il faudra au préalable enrichir le fichier `entree.dimacs` d'une nouvelle ligne, pour éliminer l'affectation qu'on vient d'afficher).

Les points (2), (3) et (4) ci-dessus décrivent une boucle.

Il sera utile de sauvegarder dans une référence vers une liste de formules (clauses disjontives), les clauses qui expriment la condition de stabilité. En effet, cette liste de clauses doit évoluer au fur et à mesure des générations stables engendrées.

3 Extensions

Toutes les extensions seront les bienvenues, et leur présence sera prise en compte lors de l'évaluation.

Interface graphique. Une extension plutôt simple à mettre en œuvre et donnant une certaine satisfaction est bien sûr l'implémentation d'une interface graphique.

Les fonctionnalités d'une telle interface pourront être : (i) la possibilité de rentrer une génération initiale en cliquant avec la souris les cases d'une grille vide, affichée dans une fenêtre ; (ii) étant donné un automate et une génération initiale, afficher l'évolution du système, en montrant les générations obtenues les unes après les autres selon un certain intervalle temporel ; (iii) étant donné un automate, afficher dans une fenêtre les générations stables par le biais de `minisat`.

La bibliothèque standard d'Ocaml fournit dans le module `Graphics` des fonctionnalités de base qui devraient suffire pour développer une telle interface.

Générations périodiques. En plus des générations stables, on pourrait s'intéresser à l'étude des générations périodiques (ou *oscillateurs*), qui reviennent à leur état d'origine au bout d'un nombre fini de générations. Plus précisément, une génération G est dite *périodique d'ordre n pour un automate C* s'il existe $n + 2$ générations $G_0, \ldots, G_{n+1}$ telles que : $G_0 = G = G_{n+1}$ et, pour tout $i \leq n$, l'automate C fait évoluer G_i en G_{i+1}. Remarquez que les générations stables sont les générations périodiques d'ordre 0.

On pourra donc imaginer d'utiliser `minisat` pas seulement pour calculer les générations stables, mais aussi les générations périodiques pour un ordre donné. Ce problème n'est par trivial. En particulier, il faut trouver la formule caractérisant les générations stables d'ordre n (ou

d'ordre au plus n). Attention, le nombre de contraintes peut exploser quand la période grandit. Il sera donc sage de se restreindre à des petits nombres comme 1 ou 2.

Variantes d'automate cellulaire. Il y a une myriade de définitions d'automates cellulaires sur une grille bidimensionelle. Nous nous sommes restreints aux automates à deux états et à un voisinage de Von Neumann de rayon 1.

Les cellules peuvent avoir d'autres états que A et D. Graphiquement, ces états peuvent être représentés par des couleurs différentes, ce qui donnera l'occasion d'observer des évolutions polychromatiques.

On peut utiliser des notions différentes de voisinage (voir Figure 2). Celui que nous avons considéré est de *rayon* 1, car il depends des cellules de distance au plus 1 sur la même colonne ou la même ligne de la cellule au centre du voisinage. On peut considerer de rayons plus grands, par exemple examiner 9 cellules pour le voisinage de Von Neumann de rayon 2, ou 13 cellules pour celui de rayon 3, etc. On peut aussi considerer les cellules sur les deux diagonales, on parlera alors de *voisinage de Moore*, qui est celui utilisé par Conway pour definir le *jeu de la vie* (Exemple 1).

4 Soumissions

Le projet sera à traiter en groupes de 2 personnes au plus (les projets soumis par des groupes de 3 personnes ou plus ne seront pas acceptés).

Votre solution devra consister en :

— un programme écrit en OCaml et utilisable sous Linux et/ou sur les ordinateurs de l'UFR ;

— un fichier texte nommé README contenant vos noms et indiquant brièvement comment compiler et utiliser votre programme ;

— un rapport au format PDF de 1 à 4 pages environ, décrivant sommairement votre soumission, les extensions traitées, et expliquant (ou justifiant) les choix de conception ou d'implémentation que nous pourrions ne pas comprendre du premier coup ;

— tout autre fichier nécessaire à la compilation et à l'exécution, comme des fichiers d'exemples, un fichier `Makefile`, un script permettant de compiler vos sources, etc.

Votre rendu devra consister en une seule archive compressée `tar.gz`. L'archive devra *obligatoirement* s'appeler *`nom1-nom2`*`.tar.gz`, et s'extraire dans un répertoire *`nom1-nom2/`*, où *`nom1`* et *`nom2`* sont les noms des deux personnes constituant le groupe. Par exemple, si vous vous appelez Pierre Corneille et Jean Racine, votre archive devra s'appeler `corneille-racine.tar.gz` et s'extraire dans un répertoire `corneille-racine/`.

La date limite de soumission est le mercredi 30 décembre à 23h59.

La soumission sera faite à travers la page Didel du cours. Attention, il s'agit d'un projet, pas d'un TP — vous devez donc nous fournir un programme qui fait quelque chose. Il vaut mieux faire une partie du projet qui fonctionne, que d'essayer de tout faire et nous fournir un programme où rien ne fonctionne.