Projet MIM1 : graphes d'automates cellulaires

Laure Danthony et Christophe Delage

mai 2001

1 Objectifs du projet

Les objectifs du projet mené depuis janvier 2001 étaient les suivants :

- La réalisation d'une interface utilisateur agréable pour entrer les caractéristiques de son graphe, le calcul de positions des nœuds avec possibilité de les recalculer si le résultat ne convient pas.
- La visualisation graphique de transitions d'automates cellulaires de dimension supérieure ou égale à 2.
- Le calcul des transitions d'automates cellulaires plus complexes, qui réalisent par exemple un décompte des nœuds du graphe.

Ce projet a principalement été réalisé dans les langages *C* et *OCaml*. L'utilisation de ces deux langages pour produire un seul exécutable est rendu possible par *OCaml* : il est capable d'appeller des fonctions *C* (étant lui-même écrit en *C*) et il fournit une interface de programmation pour accéder à la structure des variables *Caml*.

Dans les sections suivantes, nous montrerons comment nous avons séparé les différentes tâches en modules, et pointerons les principales difficultés rencontrées.

2 Affichage esthétique d'un graphe

Ce premier objectif qui n'est qu'une étape vers la visualisation d'automates cellulaires à plusieurs dimensions nous a pris plus de temps, notamment pour des aspects esthétiques. Le critère de choix pour la dernière version était le graphe de Petersen : une des itérations de notre affichage donnaît un dessin que nous avons jugé assez "joli" pour passer à la phase suivante de notre projet.

2.1 Le calcul des positions

Ce calcul a été l'occasion de refaire un peu de mécanique. Pour afficher un graphe en effet, il faut :

- que les nœuds soient les plus écartés possibles ;
- que les arêtes prennent le plus d'espace disponible ;
- que les arêtes se croisent le moins possible.

Nous avons donc reconsidéré le problème de calcul des positions des nœuds du graphe d'une façon plus physique :

- les nœuds sont considérés comme des points matériels qui se repoussent ;
- les arêtes sont des ressorts qui maintiennent de la rigidité entre les nœuds ;
- le tout se meut dans un liquide (frottement fluide) ;

– trouver une solution stable (à un coefficient ε près).

Nous avons réalisé cette partie de calcul des positions (dans le plan complexe) dans le langage C qui permet des calculs plus rapide et aussi une meilleure gestion de la précision. Le code se trouve dans les fichiers suivants :

- `complex.c` et `complex.h` offrent des outils de manipulation des nombres complexes (addition, norme, ...) ;
- `positions.c` réalisent le calcul en lui même, c'est à dire qu'il recherche une position stable (solution d'équations différentielles) à ε-près, à partir d'une disposition aléatoire (i.e. à permutation-près) des nœuds sur un cercle. Noter la présence de nombreuses constantes qui permettent de régler des paramètres du calcul, comme par exemple la viscosité du fluide. Ce fichier permet de calculer les positions du graphe donné afin qu'il remplisse au mieux la fenêtre graphique dont la taille est aussi donnée.

Il faut enfin noter la présence d'objets de type `value` qui est le type *C* des objets *OCaml*. On passe par exemple un objet *OCaml* de type `graph` au fichier de calcul des positions par l'intermédiaire de ce type `value`, et on peut accéder aux champs du type par la macro `Field`.

2.2 Le type graph

Nous avons choisi de définir en *Caml* le type `graph` de la façon suivante :

```
type graph = { tab : int array array;
               mutable pos : (int * int) array
             }
```

Ainsi, le type `graph` contient non seulement la matrice des arêtes existantes (à la i-ème ligne, on a le tableau des voisins du nœud i (de 0 à $n-1$)) mais aussi la position de ses nœuds dans la fenêtre graphique.

Le fichier `graph.ml` contient la déclaration du type `graph`, divers outils qui permettent sa modification, ainsi que l'appel externe à la fonction `c_positions` qui modifie le champ `pos` du graphe, si on lui passe en paramètre la dimension de la fenêtre graphique ouverte. La fonction `recomputepos` permet un re-calcul des positions du graphe, elle sera accessible à l'utilisateur pour qu'il puisse chercher un dessin convenable du graphe.

2.3 L'affichage graphique

Les fonctions réalisant l'affichage graphique d'un graphe dont le champ `pos` contient un calcul correct des positions se trouvent à l'intérieur des fichiers `gdrawing.ml` et `gdrawing.mli`. Nous avons aussi écrit une fonction qui exporte le dessin d'un graphe au format EPS (Encapsuled Post-Script) ce qui permet d'inclure ces images directement dans un document (par exemple celui-ci !). Ces fonctions se passent de commentaire : ce sont principalement des allocations de couleurs aux nœuds et aux arêtes. Noter l'utilisation bien pratique de la fonction `Array.iteri` pour appliquer la même fonction sur tous les nœuds du graphe.

Voilà ce qu'il est possible d'obtenir à l'aide du module de dessin de graphes :

2.4 Construction de graphes à partir d'un fichier ou de l'entrée standard

Il s'agit ici de coder un type graphe dans un fichier ou à partir de l'entrée standard, et de réaliser un "convertisseur" qui permettra de construire un objet de type `graph`(*Ocaml*) à partir du fichier ou du flux. Ces convertisseurs sont réalisés à partir des outils de lexeur (*ocamllex*) et de parseur (*ocamlyacc*) de *OCaml* : voir les fichiers `glexer.mll`, `gparser.mly`, `gio.ml` et `gio.mli`.

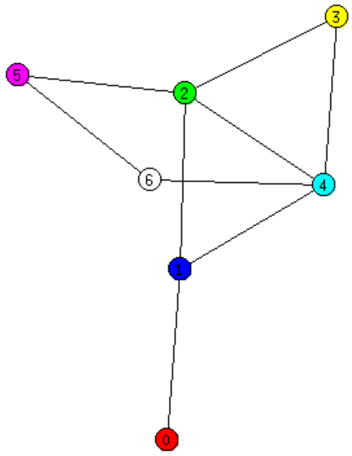

FIG. 1: *Un graphe quelconque*

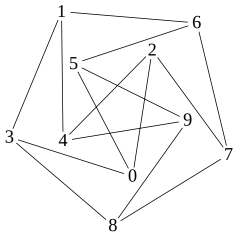

FIG. 2: *Le graphe de Petersen*

- **A partir d'un fichier :** le codage choisi est du type :

```
4 2
# 1
0 2
1 3
2 #
```

  Ce fichier désigne le graphe à 4 sommets, de degré maximum 2, tel que : le premier voisin du nœud 0 est #, le deuxième est 1 (0 n'a en fait qu'un seul voisin), 1 a pour voisins 0 et 2, etc. La fonction `file_read` (qui sera rendue accessible à l'utilisateur) réalise la conversion vers `graph`.

- **A partir de l'entrée standard :** la syntaxe choisie est la suivante :

```
0-1 1-2 2-3 3-4 4-5 5-6 6-0
```

  on choisit de ne marquer que les arêtes existantes, ce qui est plus simple pour l'utilisateur. Notons que l'on ne donne pas le degré maximal du graphe (ce n'est pas très important de toute façon). La conversion se fait à l'aide de la fonction `read`.

2.5 Remarques sur la compilation

Nous nous sommes (fortement) inspiré du Makefile de Nachos (un système écrit en C++) pour réaliser un Makefile automatique (qui gère les dépendances entre les fichiers sources).

3 Visualisation des transitions d'automates cellulaires à plusieurs dimensions

Nous passons donc maintenant aux automates cellulaires. Nous allons tout d'abord rappeler leur définition, puis montrer comment nous avons géré les transitions et leur affichage .

3.1 Qu'est-ce qu'un automate cellulaire?

Dans cette section, nous introduisons les automates cellulaires "synchrones" de façon assez informelle, mais cette définition nous sera suffisante pour nous permettre de les visualiser.

Pour définir un automate cellulaire à d-dimensions sur un graphe à n noeuds, il faut tout d'abord que chaque sommet du graphe ait un degré égal à d (sinon, on ajoute des nœuds fictifs, "dièse"), et il faut se donner un ensemble d'états de ces nœuds, Q. Ensuite, on se donne une fonction de transition $Q \times Q^d \to Q$, c'est à dire que l'état d'un sommet à l'instant $t+1$ dépend à la fois de son état au temps t et de l'état de ses voisins au temps t. Notons que l'énumération des voisins des sommets a de l'importance pour l'application de la fonction de transition.

Une évolution de l'automate cellulaire est donc totalement déterminée par les états initiaux des nœuds du graphe. Remarquons que cette évolution est cyclique, puisque le nombre d'états est une quantité finie.

3.2 Codage et visualisation des transitions

Les fichiers `cella.ml` et `cella.mli` contiennent les fonctions nécessaires pour la visualisation de l'évolution d'automates. Pour réaliser un automate, c'est-à-dire pouvoir appeler la fonction `draw_step` qui réalise un pas de calcul (modifie les états des nœuds et les redessine en une couleur dépendant de leur état), il faut coder des fonctions de type:

- (fonction de transition) `type tfun = int -> int array -> int`

- (fonction de coloration) `type colfun = int -> color`

qui sont passés en argument à la fonction `draw_step`. Ce sont ces fonctions et aussi le tableau des états initiaux qui déterminent le fonctionnement et l'affichage de l'automate cellulaire.

Notons qu'on ne peut pas associer une couleur à chaque état, car la distinction serait difficile et on ne dispose que d'un nombre restreint de couleurs "visuellement" distinctes. De plus un automate même simple contient vite un très grand nombre d'états.

A ce stade, nous disposons de tous les outils nécessaires au codage et à la visualisation de beaux graphes et d'automates cellulaires sur ces graphes. Il reste maintenant à offrir une interface utilisateur.

4 Un toplevel Caml pour l'utilisateur

Pour que l'utilisateur ait le moins possible à se plonger dans un manuel, nous voulons lui offrir une interface utilisateur agréable, qu'il puisse utiliser sans taper la moindre ligne de code. Mais si cela est possible pour afficher le graphe de son choix, il semble ardu de pouvoir décrire un automate cellulaire en "langage de tous les jours". En effet, pour un automate cellulaire de dimension 3, à 5 états , la fonction de transition est de la forme $Q \times Q^3 \to Q$, soit 5^4 transitions à coder ! L'utilisateur devra donc apprendre des rudiments du langage *OCaml* (mais qu'il ne s'inquiète pas, il aura une aide à sa disposition et aussi des exemples !).

L'exécutable fourni par la commande `make` est donc un "toplevel" *OCaml* qui, au lancement, appelle le fichier `.ocamlinit`. Celui-ci se charge du lancement du module de dessin des graphes, de construction des graphes à partir de l'entrée standard ou d'un fichier, du module de gestion des automates cellulaires, ainsi que quelques exemples de graphes et un exemple simple d'automate cellulaire. Il est possible d'avoir une aide succincte en tapant `help();;` à l'invite du programme :

```
        Objective Caml version 3.00

type "help ();;" for help.
#
```

Voilà, l'interface utilisateur est à ce stade terminée, même si elle peut encore être améliorée.

5 Deux automates particuliers

Dans cette section, nous allons illustrer la puissance des automates cellulaires en les faisant "calculer". Nous allons centrer notre étude sur des automates linéaires, mais ce n'est pas facile pour autant ! Les automates présentés savent compter et repérer les nombres premiers.

5.1 Un automate qui compte le nombre de nœuds du graphe

Étant donné un graphe linéaire à n nœuds, l'automate cellulaire que nous allons construire va afficher en binaire (vert= 1, rouge= 0, les bits de poids faible à droite) ce nombre n.

Nous séparons le problème en deux :

1. faire en sorte que le nœud de droite reçoive n signaux de présence ;

2. réaliser le compteur binaire qui sera incrémenté chaque fois qu'un signal de présence sera reçu.

On note g et d les voisins de gauche et de droite, ainsi que k_g (resp k, k_d) le bit k du voisin de gauche (resp. du nœud courant et du voisin de droite). Pour des raisons de clarté, on distingue k qui fait référence à l'état avant la transition de k' qui indique l'état après la transition. Par commodité, on considère que k_g est nul si $g = \#$ (idem pour k_d).

Nous avons besoin de deux bits pour envoyer les signaux de présence : le bit du signal lui-même (p), et un bit pour le signal qui va provoquer l'envoi du premier (s). Le signal s doit se déplacer vers la gauche et ne doit être envoyé qu'une fois, p doit être envoyé quand s passe, sinon il est transmis vers la droite.

Pour réaliser un compteur binaire, il nous faut deux autres bits (un pour le bit du compteur (c), et un pour le signal d'incrément (i)). i doit se déplacer vers la gauche, s'il rencontre un bit à un il met zéro et continue, si le bit est a zéro il le met à un et s'arrête, de plus on fait partir i chaque fois que p arrive sur le nœud de droite.

En résumé :

$$\begin{array}{ll}
\text{si } d = \# & s' \leftarrow 0 \\
 & p' \leftarrow s \vee p_g \\
 & i' \leftarrow p \\
 & c' \leftarrow c \text{ xor } i \\
\text{sinon} & s' \leftarrow s_d \\
 & p' \leftarrow s \vee p_g \\
 & i' \leftarrow i_d \wedge c_d \\
 & c' \leftarrow c \text{ xor } i
\end{array}$$

L'état initial est :

- pour le nœud de droite, $s = 1, \quad p = i = c = 0$;
- pour les autres nœuds, $s = p = i = c$.

Le code de cet automate (certes un peu abscons) est disponible dans les fichiers `ca_count.ml` et `ca_count2.ml` pour la version optimisée. L'utilisateur peut visualiser le travail de cet automate en faisant par exemple :

```
# #use "ca_count.ml" ;;
# speed := 0.5;;
# run 15;;
```

Admirez le résultat :

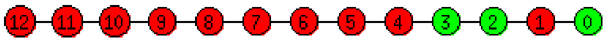

FIG. 3: *Il compte en couleurs !*

5.2 Un automate qui repère les nombres premiers

L'astuce consiste bien évidemment à utiliser la méthode ancestrale du crible d'Érathostène, mais comment? La principale difficulté est que le nombre d'états que l'on dispose est fixé à l'avance, et donc est forcément borné.

Pour réaliser le crible d'Erathostène, il faut pouvoir marquer les nœuds qui sont multiples de k (arbitraire). L'idée est d'utiliser principalement deux signaux dans un schéma de ping-pong (une "raquette" et une "balle") et une "table" comme sur la figure 4. Il est clair que les signaux "balle" et "raquette" se croisent exactement aux multiples de k, on marque donc les intersections comme "non premiers". Apellons ce mécanisme une *passe*.

Le fonctionnement général de l'automate est (les nœuds sont numérotés de 1 à n) :

1. au début, le nœud 1 est marqué "non premier", tous les autres sont "inconnus" ;
2. on fait partir du premier nœud un signal qui trouve le premier nœud "inconnu", on marque ce nœud "premier" et on en fait partir une *passe* ;

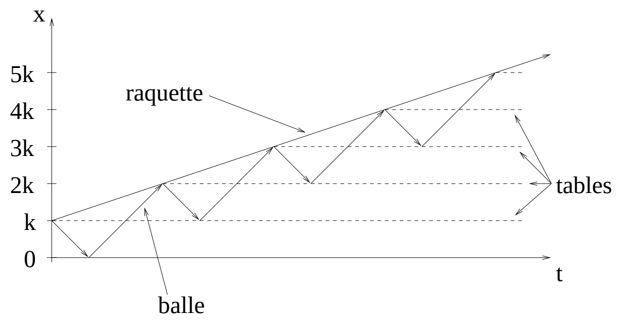

FIG. 4: *Tennis de table*

3. lorsque la “raquette” a atteint le $n^{\text{ème}}$ nœud, on renvoie un signal qui, arrivé au premier nœud, refera l'étape 2 ;

4. si le signal du 2 arrive au bout du graphe, on a fini.

La fonction de transition ne peut être détaillée ici par manque de place (elle prendrait deux pages à elle seule !).

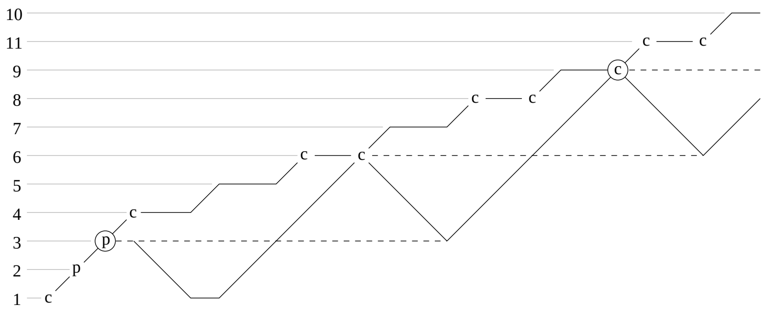

FIG. 5: *L'automate ca_prim en action*

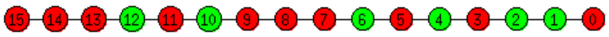

FIG. 6: *En vrai*

6 Conclusions et objectifs

Ainsi ce projet a été l'occasion de découvrir l'univers très particulier des automates cellulaires et de se confronter aux difficultés de codage de fonctions de transitions. Nous avons vu que même pour des automates cellulaires linéaires, les fonctions de transition se codent en plusieurs pages. Alors pour des graphes quelconques !

Par ailleurs, une extension possible à notre projet était de faire un automate qui “linéarise” un graphe quelconque (en faisant un parcours en profondeur : on considère que les voisins d'un nœud dans le nouveau graphe sont les antécédent et suivant dans l'ordre du parcours), et ensuite “d'interfacer” cet automate avec ceux décrits dans la section précédente. Malheureusement, cela pose de gros problèmes pour la transmission des signaux car deux “voisins” dans le parcours ne sont pas forcément voisins dans le graphe, la synchronisation le l'automate simulé devient alors un vrai casse-tête.