Automates cellulaires en dimension 1

Adapté d'un sujet de TP d'agrégation

1 Automates cellulaires

Un automate cellulaire est un système comprenant des cellules réparties régulièrement. Chaque cellule est décrite par un état, **actif** (`true` ou `1`) ou **inactif** (`false` ou `0`). L'évolution dans le temps de l'état de chaque cellule ne dépend que de l'état de ses cellules voisines. Une règle, identique pour toutes les cellules qui constituent l'automate, définit cette évolution.

Le cas le plus simple, et qui sera étudié ici, est le cas appelé 1D. Il s'agit d'une ligne de cellules, chaque cellule ayant deux voisins.

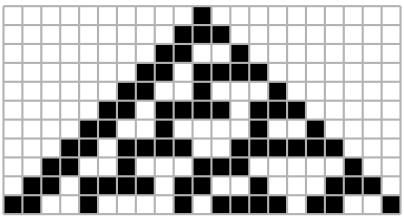
On considère le temps comme étant discret, et l'état d'une cellule au temps t dépend uniquement de son état et de celui de ses voisines au temps $t-1$. On appelle une **génération** l'état de l'ensemble des cellules à un tel instant t.

On peut ainsi définir de manière univoque l'évolution d'un tel système en utilisant uniquement l'état initial du système et la règle permettant en fonction du voisinage d'obtenir le nouvel état. On appellera cette règle **règle d'évolution**.

Par exemple, la table suivante donne un exemple de règle :

Motif initial (instant t)	000	001	010	011	100	101	110	111
État suivant de la cellule centrale (instant $t+1$)	0	1	1	1	1	0	0	0

Dans le cas où une cellule dans l'état 1 est entourée de deux 0, elle sera toujours dans l'état 1 à la génération suivante, d'après la règle $010 \to 1$. Dans le cas où ce 1 est entouré de deux 1, la cellule sera dans l'état 0 à la génération suivante, d'après la règle $111 \to 0$. L'utilisation de cette règle sur une ligne de cellules ne contenant initialement qu'une seule cellule dans l'état 1 donnera l'évolution suivante (chaque ligne est une nouvelle génération, de haut en bas, les cellules actives sont représentées en noir).



On considèrera dans cette étude, une ligne de départ de taille fixe, une règle et un nombre de générations. L'objectif est d'explorer plusieurs approches pour calculer l'évolution d'un automate cellulaire. On réfléchira à la performance de chaque approche.

2 Approche en OCaml

Un automate cellulaire borné est la donnée d'un ruban délimité par ses extrémités et d'une règle d'évolution. On utilisera les types suivants :

```
type ruban = bool list

type regle = bool -> bool -> bool -> bool
```

Lorsqu'on souhaite appliquer la règle d'évolution sur une des extrémités du ruban (premier et dernier éléments de la liste), on considèrera que les cellules en dehors du ruban sont dans l'état `false`.

Voici deux exemples supplémentaires de fonctions d'évolution permettant d'illustrer votre travail, mais d'autres exemples sont bienvenus :

- `evol a b c` $= a \oplus b \oplus c$ où $\oplus$ désigne le *ou-exclusif* ;
- `evol a b c` $= a \vee b \vee c$.

2.1 Une première implémentation

Question 1 Écrire une fonction `generation : ruban -> regle -> ruban` qui prend en argument un ruban et une règle d'évolution et renvoie un nouveau ruban dans la configuration obtenue après applications de la règle d'évolution.

Question 2 Écrire une fonction `afficher : ruban -> unit` qui fait un affichage (graphique ou textuel) du ruban d'un automate cellulaire. Pour un affichage textuel, on choisira des caractères permettant de différencier facilement les cellules actives et inactives.

Question 3 En déduire une fonction `transitions : ruban -> regle -> int -> ruban` qui prend en arguments un ruban, une règle d'évolution et un entier n et renvoie l'état du ruban après n générations. La fonction fera un affichage du ruban à chaque génération.

2.2 En utilisant un transducteur

Un **transducteur** est un automate fini déterministe qui écrit un mot en sortie lors de la lecture d'un mot en entrée. Depuis un état donné, lors de la lecture d'une lettre, le transducteur permet de :

- écrit une lettre en sortie, en fonction de l'état courant et de la lettre lue ;
- détermine le nouvel état atteint, en fonction de l'état courant et de la lettre lue.

Il n'y a pas de notion d'état final pour un transducteur, car seul le mot écrit en sortie est important. On constate qu'avec la définition précédente, le mot en sortie aura toujours la même longueur que le mot en entrée.

Question 4 Proposer une définition formelle d'un tranducteur au sens décrit précédemment.

On propose dans cette partie de pré-calculer les transitions à l'aide d'un transducteur. Pour cela, on considère un transducteur dont les états correspondent aux mots de taille 3 sur l'alphabet $\{0, 1\}$ et chaque transition correspond à une étape de lecture du ruban. Par exemple, depuis l'état 100, si on lit un 1, on passera dans l'état 001. Ainsi, en faisant la lecture du ruban dans l'automate fini déterministe, l'état courant correspond aux trois dernières lettres lues. La fonction de sortie du transducteur dépendra de la fonction d'évolution.

Question 5 Proposer un type OCaml permettant d'implémenter un transducteur sur l'alphabet $\{0, 1\}$.

Question 6 Expliquer comment utiliser un transducteur pour calculer une nouvelle génération d'un automate cellulaire. Comment modifier le tranducteur pour calculer 2 générations d'un coup ? On commencera par déterminer le nombre d'états d'un tel transducteur.

Question 7 Écrire une fonction qui construit un transducteur permettant de calculer une génération à partir d'une règle d'évolution, puis une fonction qui calcule l'évolution d'un ruban après n générations en utilisant ce transducteur

2.3 Approche inverse

Dans les questions précédentes, on a effectué le calcul d'une nouvelle génération en fonction d'un ruban et d'une règle d'évolution donnés. Dans cette partie, on considère l'approche inverse : on dispose de deux rubans

donnés et on cherche à déterminer, si elle existe, une règle d'évolution permettant de passer du premier au second.

Question 8 Écrire une fonction `trouver_regle : ruban -> ruban -> regle option` qui prend en arguments deux rubans de même taille et renvoie `Some evol` s'il existe une règle d'évolution `evol` permettant de passer du premier au second en une génération, et `None` sinon.

On pourra proposer une ou plusieurs stratégies et discuter de leur pertinence et efficacité.

3 Approche en C

Le but de cette partie est d'évaluer de manière similaire le coût et le gain d'un pré-calcul dans l'implémentation du calcul des générations.

Comme précédemment, on considèrera un ruban limité en taille, et les cellules au delà des extrémités seront considérées comme inactives.

3.1 Implémentation naïve

On représente dans un premier temps un ruban par un (pointeur vers un) tableau de booléens `bool*`.

Question 9 Que vaut R, le nombre de règles d'évolutions différentes ? Comment utiliser un entier compris entre 0 et $R-1$ pour représenter de manière unique une règle d'évolution ? Que vaut l'entier associé à la règle donnée en exemple en introduction et que valent les entiers correspondant aux deux règles données en exemple dans la partie précédente ?

Question 10 Écrire une fonction `void transitions_naif(bool* ruban, int t, int regle, int n)` qui prend en argument un ruban, sa taille t, un entier correspondant à une règle d'évolution selon la question précédente et un nombre de générations n et modifie le ruban **en place** (donc sans créer de nouveau ruban) pour qu'il corresponde à l'évolution de la valeur initiale du ruban selon la règle, après n générations.

On pourra, comme en OCaml, écrire une fonction d'affichage du contenu du ruban.

3.2 Pré-calcul de sous-sections

Pour améliorer le calcul, on choisit de représenter un ruban comme une suite d'entiers de $[\![0, 255]\!]$ dont la concaténation des écritures binaires correspond aux états des cellules. Par exemple, la suite $(72, 194, 168)$ correspond au ruban 010010001100001010101000.

Question 11 Écrire une fonction `transitions` similaire à la question précédente, mais qui utilise ce principe pour faire les calculs de générations. On pourra supposer que le ruban a une taille qui est un multiple de 8.

Question 12 Comparer les performances des deux approches précédentes. On donne en annexe un moyen de chronométrer l'exécution d'un programme.

3.3 Implémentation concurrente

Dans le cas d'un système multi-cœur, il est possible d'accélérer encore les calculs en les répartissant pour les effectuer en parallèle.

Question 13 Adapter le code de l'implémentation naïve pour faire un calcul concurrent des générations et comparer les performances.

Annexe : rappels de programmation

Rappels en OCaml

La compilation d'un fichier `source.ml` peut se faire avec la commande `ocamlc source.ml`. Comme en C, on peut rajouter des options de compilation avec `ocamlc` *`options`* `source.ml` où *`options`* doit être remplacé par une ou plusieurs options de compilation. On cite par exemple :

- choix du nom d'exécutable : `-o nom_exec`

Rappels en C

La compilation d'un fichier `source.c` peut se faire avec la commande `gcc source.c`. On peut rajouter des options avec la commande `gcc` *`options`* `source.c` où *`options`* doit être remplacé par une ou plusieurs options de compilation. On cite par exemple :

- choix du nom d'exécutable : `-o nom_exec`
- avertissements : `-Wall`, `-Wextra`
- alerte mémoire : `-fsanitize=address`

On peut lancer un fichier exécutable `nom_exec` par la commande `./nom_exec`.

Pour utiliser les booléens, on doit inclure la bibliothèque `stdbool.h`.

Pour mesurer la performance d'un programme, on peut utiliser le code suivant :

```
struct timeval start, stop;
gettimeofday(&start, NULL);
// Partie à chronométrer
gettimeofday(&stop, NULL);
double delta = (stop.tv_sec - start.tv_sec) + 1e-6 * (stop.tv_usec - start.tv_usec);
```

Le flottant `delta` contiendra le temps, en secondes, écoulé dans la partie à chronométrer. Il faudra ajouter `#include <sys/time.h>` en entête pour utiliser ce code.

On rappelle les fonctions pour manipuler les fils d'exécution, les mutex et les sémaphore en C. Pour les deux premiers, il faut inclure la bibliothèque `pthread.h`, pour les sémaphores, il faut `semaphore.h` :

- `pthread_t fil;` pour créer un fil d'exécution ;
- `pthread_create(&fil, NULL, f, arg);` pour lancer un fil d'exécution qui calcule `f(arg)`. `f` doit être une fonction qui prend en argument un pointeur et renvoie un pointeur (généralement `NULL`), `arg` doit être un pointeur. Dans la fonction `f`, on doit convertir ce pointeur en lui redonnant le bon type si nécessaire (car il sera transtypé en `void` lors de l'appel) ;
- `pthread_join(fil, NULL);` attend la fin de l'exécution d'un fil ;
- `pthread_mutex_t verrou;` pour créer un mutex ;
- `pthread_mutex_init(&verrou, NULL);` pour initialiser un mutex ;
- `pthread_mutex_destroy(&verrou);` pour détruire (libérer la mémoire allouée) un mutex ;
- `pthread_mutex_lock(&verrou);` pour verrouiller un mutex ;
- `pthread_mutex_unlock(&verrou);` pour déverrouiller un mutex ;
- `sem_t sem;` pour créer un sémaphore ;
- `sem_init(&sem, 0, k);` pour initialiser un sémaphore avec un compteur égal à k ;
- `sem_destroy(&sem);` pour détruire un sémaphore ;
- `sem_wait(&sem);` pour attendre (décrémenter) un sémaphore ;
- `sem_post(&sem);` pour libérer (incrémenter) un sémaphore.