TP4: langage C

David Baelde

7 et 8 octobre 2015

Dans ce premier TP de C, nous allons utiliser quelques types de base, des tableaux et un peu de pointeurs. Nous éviterons pour cette fois l'allocation dynamique de mémoire. On évite ainsi certaines difficultés du langage, et on peut déja faire des choses amusantes.

Exercice - 1 *La politesse*

1- De tête, que fait le programme suivant ?

```
#include <stdio.h>
void affiche(char* message) {
  printf("> %s <\n",message);
}
int main() {
  char m[] = {'H','e','l','l','o','!','\0'};
  char* s = m;
  int i;
  for (i=0; i<7; i++)
    affiche(s++);
  return 0;
}
```

2- Écrivez ce programme dans le fichier `hello.c`, testez :

```
$ gcc hello.c -o hello && ./hello
```

3- Et le programme suivant ?

```
#include <stdio.h>
char* hello() {
  char m[] = {'H','e','l','l','o','!','\0'};
  char* s = m;
  return s;
}
int main() {
  printf("> %s <\n",hello());
  return 0;
}
```

Exercice - 2 *Automates cellulaires*

On considère des automates cellulaires uni-dimensionnels à états booléens. Concrètement, on s'intéresse à un système de "cellules" organisé selon une ligne bi-infinie orientée, où chaque cellule a un voisin à gauche et un voisin à droite, et est dans un état pris dans $\{0, 1\}$. L'évolution du système est synchrone et locale : à chaque étape, toutes les cellules changent d'état en même temps, en fonction de son propre état et de ceux de

leurs deux voisins immédiats. Si s, s_g et s_d sont respectivement les états d'une cellule, de sa voisine de gauche et de celle de droite, le prochain état de s est donné par $f(s_g, s, s_d)$.

Par exemple, si $f(x, y, z) = 1$ ssi $x + y + z = 1$, alors on a la transition suivante :

t	…	0	0	1	1	…
$t+1$	…	?	1	0	?	…

La fonction f est la table de transition ou règle de calcul de l'automate. Elle a $2^3 = 8$ entrées possibles, et renvoie un booléen : on peut donc coder ces fonctions sur un octet (et pas moins). Précisemment, si n est un entier 8 bits, la fonction associée sera celle qui à (s_g, s, s_d) associe le bit de poids 2^k de n, où k s'écrit $s_g s s_d$ en binaire, avec s_g bit de poids fort. Cette notation a été popularisée par le mathématicien Wolfram, qui a étudié les diverses règles possibles en fonction du type de comportements qu'elles engendrent.

1- Ecrivez la fonction qui calcule la fonction de transition associée à un octet. Son prototype devra être le suivant :

```
bool transition(char rule, bool left, bool here, bool right);
```

Pour utiliser le type `bool`, qui n'est pas prédéfini en C, il faudra inclure l'en-tête `stdbool.h`. Pour tester, on incluera les lignes suivantes dans la fonction `main`, après avoir inclus l'en-tête `assert.h` (il déclare la fonction `assert()`, qui sert à indiquer une erreur à l'exécution quand son argument est faux) :

```
assert(transition(4,false,true,false));
assert(transition(7,false,true,false));
assert(transition(64+4,true,true,false));
assert(!transition(64+4,true,true,true));
assert(transition(128,true,true,true));
assert(!transition(128,true,true,false));
```

2- Nous allons simuler des automates sur un anneau de taille N. La taille sera fixée à la compilation, ce qui permettra de n'utiliser que des tableaux alloués statiquement. Pour pouvoir changer la valeur de N facilement, nous allons cependant le définir au début du fichier par le biais d'une directive du préprocesseur :

```
#define N 16
```

Définissez ensuite une fonction qui prend un tableau de taille N représentant l'état précédent de l'automate, un autre tableau destiné à stocker l'état suivant, et remplit ce deuxième tableau selon la règle de calcul passée en premier argument. Votre fonction devra respecter le prototype suivant :

```
void step(int rule, bool prev[N], bool next[N]);
```

3- Ecrivez une fonction pour afficher un état du système, comme une ligne de caractères `'0'` et `'1'` séparés par des espaces :

```
void print_line(bool cur[N]);
```

4- Dans votre fonction `main()`, déclarez deux tableaux de N booléens, initialisez le premier avec un seul 1 au centre du tableau, et testez les fonctions précédentes sur une règle simple, comme 0, 255 ou 1.

5- Modifiez votre fonction principale pour utiliser comme règle un entier fourni par l'utilisateur en premier argument sur la ligne de commande. Pour cela on donnera à `main` son prototype général :

```
int main(int argc, char** argv);
```

Le premier argument donne le nombre de paramètres sur la ligne de commande, en prenant en compte le nom du programme lui-même. Le second argument est un tableau de chaînes de caractères, de taille `argc`. Par exemple, si on tape dans le shell `./monprog` alors `argc` vaudra 1 et `argv` sera `{"./monprog"}` et si l'on tape `./monprog 12` alors on aura `argc==2` et `argv` vaudra `{"./monprog","12"}`. Afin de convertir une chaîne en entier, on utilisera `atoi`. Que se passe-t-il quand votre programme est lancé sans argument sur la ligne de commande ? Essayez de gérer "gracieusement" ce cas, en informant l'utilisateur du bon usage de votre programme.

6- Modifiez votre fonction principale pour afficher N états successifs de votre automate. Pour cela, on utilisera seulement deux tableaux, en utilisant la fonction `transition()` pour calculer alternativement l'un à partir de l'autre et l'autre à partir de l'un.

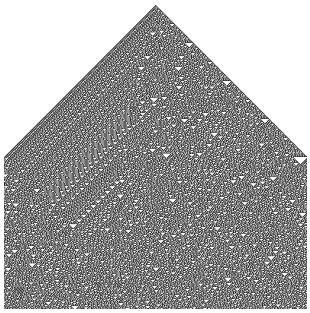
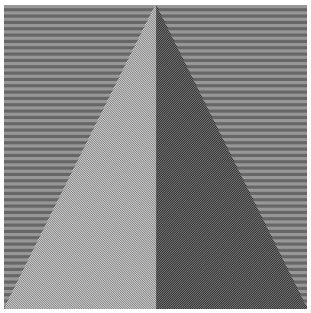
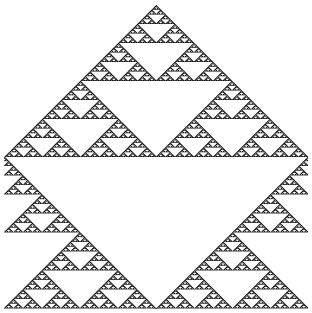
Exercice - 3 *Diagrammes espace-temps*

Ce qui est bien avec les automates 1D, c'est qu'on peut visualiser en 2D leur diagramme espace-temps : il s'agit bêtement de la suite de lignes que votre programme calcule. Mais pour y voir quelquechose d'intéressant, il faut augmenter N, et pour que cela soit joli on va produire une image.

1- Avant vos N lignes de N `'0'` et `'1'` séparés par des espaces, affichez `"P1"` sur une première ligne, puis deux fois N (en notation décimale usuelle) séparés par un espace sur une deuxième ligne. Vous avez produit un fichier au format *plain* pbm, que vous pouvez visualiser avec un afficheur d'image quelconque. (La seconde ligne indique simplement les dimensions de l'image, et la suite indique la succession de pixels noirs et blancs.)

```
$ ./automate 12 > 12.pbm # produire l'image
$ head -n 2 12.pbm       # verifier l'en-tete
P1
16 16
$ ristretto 12.pbm       # afficher l'image
```

C'est l'occasion d'augmenter la taille de votre simulation, et visualiser les diagrammes obtenus pour différentes règles. Ci-dessous, quelques exemples de ce qu'on peut trouver.

Le format d'image *plain* pbm, basé sur la notation ASCII, a l'avantage d'être facile à produire, et facile à lire pour un être humain. Mais les fichiers obtenus sont inutilement gros : un bit d'information est codé sur deux octets. Le format d'image pbm bit à bit améliore ceci. Il s'obtient en remplaçant l'en-tête P1 par P4, en laissant la seconde ligne inchangée, et en remplaçant les lignes suivantes par un codage au niveau du bit.

Chaque ligne est codée comme une succession d'octets. Les pixels de la ligne, lus de gauche à droite, sont codés comme les bits de la suite d'octets, lus de gauche à droite et dont les bits sont lus du poids fort au poids faible. (En général, le dernier octet peut contenir des bits non significatifs, si la longueur des lignes n'est pas un multiple de huit, mais on pourra prendre cette hypothèse simplificatrice ici.) Les lignes sont simplement concaténées, sans caractère de retour à la ligne entre deux lignes.

2- Adaptez votre programme pour produire des images au format pbm compact.

Exercices bonus, au choix

Exercice - 4 *Puissance 4*

Une grille de puissance 4 est composée de 6 lignes et 7 colonnes. On se propose de représenter les positions d'un joueur de puissance 4 sur 32 bits de la façon suivante :

```
.  .                 .
5 12               47
4 11   etc.        46
3 10       etc.    45
2  9               44
1  8 15 22         43
0  7 14 21 28 35 42
```

On indique ici, à une position dans la grille, legg poids du bit utilisé pour coder cette position. On notera que les bits de poids $7k - 1$ (notés par des `.` ci-dessus) ne sont pas utilisés : on devra laisser ces bits à zéro.

On commencera par poser la définition suivante, s'appuyant sur le fait qu'un entier `long long` est codé sur au moins 64 bits — ce qu'on pourra vérifier en affichant ou testant la valeur de `typeof(board)` :

```
typedef
  unsigned long long
  board;
```

1- Comment tester l'existence d'un alignement vertical de quatre positions dans un `board` ?

2- Comment tester l'existence d'un alignement horizontal de quatre positions dans un `board` ?

3- Comment tester l'existence d'un alignement diagonal de quatre positions dans un `board` ?

4- Coder une fonction indiquant si un joueur est gagnant, étant donné le `board` représentant ses positions.

5- Coder un jeu de puissance 4 interactif. On pourra utiliser la fonction scanf pour lire un caractère sur la ligne de commande — documentation sur le web ou `man 3 scanf`.

6- Pour aller plus loin…

- Coder un algorithme min-max pour le puissance 4. La fonction prendra une borne sur la longueur des branches à explorer. Pour évaluer les feuilles de l'arbre exploré, dans le cas d'une paire de grille où personne ne gagne, on pourra prendre la différence des nombre d'alignements de taille 3 des deux joueurs.
- Adaptez votre fonction pour implémenter l'algorithme alpha-beta.
- Exploitez les résultats de John Tromp, qui a "résolu" le jeu : `http://tromp.github.io/c4/c4.html`. On trouve notamment sur cette page un fichier contenant l'évaluation parfaite (quel joueur gagne) de toutes les positions du jeu après 8 coups.

Exercice - 5 *Réaction-Diffusion*

Alan Turing a montré qu'on pouvait expliquer des phénomènes biologiques variés (par exemple, les rayures d'un zèbre, d'un poisson, ou les premières étapes du développement d'un embryon) en considérant les états stables de systèmes où plusieurs composés chimiques réagissent entre eux et diffusent dans un milieu. Il a modélisé mathématiquement ces systèmes chimiques, et déterminé ainsi la forme des états stables dans certains cas. Il a aussi simulé ces modèles sur ordinateur, pour confirmer ses résultats et tenter de faire des prédictions dans des cas plus complexes.

Dans cet exercice, on considère la simulation d'un système à deux composés, sur un tore discrétisé : concrètement, une matrice $N \times N$ dont on a recollé les bords gauche et droit, ainsi que haut et bas. Nos composés, appelés A et B, évoluent selon les équations suivantes [1] :

$$\frac{\Delta A}{\Delta t} = D_A \nabla^2 A - AB^2 + f(1 - A) \qquad \frac{\Delta B}{\Delta t} = D_B \nabla^2 B + AB^2 - (k + f)B$$

1. Référence, avec schémas, explications, et vidéos : `http://karlsims.com/rd.html`

Le premier terme de chaque équation représente le coefficient de diffusion, proportionnel à une constante D_A (resp. D_B) et au gradient du composé concerné. Ce gradient sera calculé concrètement ainsi sur notre grille :

$$\begin{aligned}(\nabla A)(i,j) \quad = \quad & 0.2 \times \big(A(i,j+1) + A(i+1,j) + A(i-1,j) + A(i,j-1)\big) + \\ & 0.05 \times \big(A(i+1,j+1) + A(i+1,j-1) + A(i-1,j+1) + A(i-1,j-1)\big) \\ & -A(i,j)\end{aligned}$$

Le second terme correspond à la réaction : ici, deux B et un A réagissent pour faire un B, en consommant le A mais pas les B. (On peut penser à des lapins qui se reproduisent en consommant de la nourriture…)

Le dernier terme est un apport extérieur au système : les B meurent spontanément, proportionnellement à leur population ; les A sont apportés spontanément au système.

On prendra [2] les constantes suivantes : $D_A = 1$, $D_B = 0.5$, $f = 0.062$ et $k = 0.0609$.

1- Implémenter la fonction `nabla` codant le gradient comme indiqué ci-dessus.

2- Implémenter la fonction de transition : elle prendra deux matrices codant l'état courant, et calculera à partir de là le nouvel état, le stockant dans une seconde paire de matrices. La cellule $A'(i,j)$ après un instant est obtenue en ajoutant à $A(i,j)$ la quantité définie par l'équation ci-dessus. (Autrement dit on simule bêtement l'équa-diff, en prenant $\Delta t = 1$.) De même pour B sauf qu'il faudra ici prendre garde à borner le résultat par 1, afin de garder toutes les quantités dans l'intervalle $[0;1]$. Vérifier rapidement que la fonction ne plante pas quand on l'exécute sur un système quelconque, par exemple où $A = 1$ partout et $B = 0$ partout.

3- Implémenter une fonction qui produit une image à partir d'un état du système. On pourra utiliser un des formats PGM (cf. `http://netpbm.sf.net/doc/pgm.html`) faits pour les images en niveaux de gris, soit en bitmap ("P5") soit en ascii ("P2").

4- Initialiser un état avec quelques régions simples (disques ou cercles) où l'on met $B = 1$. Simuler pendant 100 étapes, produire l'image résultante.

5- Adaptez votre code pour produire une image tous les 100 pas de simulation, dans un fichier contenant un numéro de série, par exemple `sim-001.pgm, sim-002`…

Pour la manipulation des fichiers, on utilisera `fopen, fwrite, fclose`.

Pour aller plus loin, après quelques cours de programmation système, on chercherait naturellement à paralléliser ce code : ce type d'algorithme s'y prête très bien.

2. Référence : `http://mrob.com/pub/comp/xmorphia/uskate-world.html`