Automates bi-dimensionnels

1 Introduction

Depuis la ligne de commande, naviguez vers un répertoire vous appartenant, et exécutez la commande suivante, qui téléchargera et décompressera un répertoire nommé `automates_2D` contenant quelques sources C et un makefile :

```
curl cdn.sci-phy.org/mp2i/tp5-automates_2D.tgz | tar xvz
```

Le fichier à compléter est le fichier `automates_2D.c`. Les fichiers `disp.h` et `disp.c` fournissent des fonctions facilitant l'affichage des automates, et le fichier `loop.h` des tableaux qui seront utiles pour le dernier automate. Les fichiers `basic_ppm.h` et `basic_ppm.c` ne sont pas nécessaires pour ce TP, mais sont fournis si vous souhaitez enregistrer le résultat d'un automate sous forme d'une image. Un fichier Python est également fourni pour faciliter la visualisation des résultats.

2 Automates à agent

2.1 Principe

Durant toute cette séance, nous nous intéresserons à des automates 2D. Ils opèrent sur une grille de taille de hauteur `nbl` et de largeur `nbc` où chacune des cases peut contenir un entier entre 0 et $p-1$. Le contenu de cette grille sera mémorisé sous la forme d'un tableau. Compte tenu des limitations concernant les tableaux multi-dimensionnels en C [1], il s'agira d'un tableau *unidimensionnel* de taille `nbl`×`nbc`. Les `nbc` premières cases contiendront les valeurs de la première ligne de la grille (de gauche à droite), les `nbc` suivantes celles de la seconde ligne, et ainsi de suite.

Comme souvent lorsque l'on travaille avec une grille, la question des bords se pose. Nous travaillerons durant ce TP avec des conditions aux bords *périodiques*, c'est-à-dire que la case « à droite » de la case située la plus à droite sur une ligne est la case se trouvant à l'autre bout, le plus à gauche, sur la même ligne, et inversement. De même, les cases de la première ligne et celles de la dernière ligne sont voisines.

Dans cette première partie, on s'intéresse à des automates faisant intervenir un « *agent* ». Il s'agit d'un objet qui se déplace dans la grille, et en fonction de son état et de l'état de la case dans laquelle il se trouve, il modifiera l'état de la case où il se trouve et déterminera le déplacement suivant. Sur une grille infinie, il s'agit en quelque sorte d'une généralisation à deux dimensions de machines de Turing, les comportements que l'on peut obtenir peuvent donc être aussi complexes qu'on le souhaite.

1. Et des suggestions du programme!

2.2 Fourmi de Langton

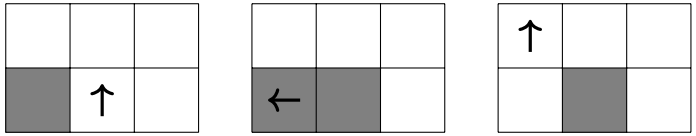
On étudie, dans un premier temps, un agent appelé *fourmi de Langton*, en hommage à son créateur Christopher Langton. On note `i` le numéro de la ligne dans laquelle elle se trouve, et `j` le numéro de la colonne. Celle-ci a également une orientation, notée `d`, prenant l'une des quatre valeurs ci-dessous :

- `d=0` : la fourmi est orientée vers la droite ;
- `d=1` : la fourmi est orientée vers le haut ;
- `d=2` : la fourmi est orientée vers la gauche ;
- `d=3` : la fourmi est orientée vers le bas.

À chaque itération, la fourmi effectue les opérations suivantes :

- si la fourmi se trouve dans une case contenant un 0, elle pivote de 90 ° vers la gauche, place un 1 dans la case, puis avance d'une case ;
- si la fourmi se trouve dans une case contenant un 1, elle pivote de 90 ° vers la droite, place un 0 dans la case, puis avance d'une case.

Voici un exemple de deux itérations de la fourmi (les cases blanches correspondent à des 0, les grises à des 1) :



1. Proposer une fonction nommée `avance_Langton`, et dont la signature serait `void avance_Langton(int tab[], int nbl, int nbc, int* i, int* j, int* d)` prenant en argument un pointeur vers un tableau, ainsi que des pointeurs vers les coordonnées et la direction de la fourmi, et effectuant une opération telle que décrite ci-dessus.

2. Pourquoi utilise-t-on des pointeurs ici?

3. Compléter la fonction `void Langton(int nbl, int nbc, int wait)` prenant en argument le nombre de lignes et de colonnes de la grille, crée avec `malloc` un tableau de la bonne taille, le remplit de zéros, place la fourmi au centre de la grille et effectue une succession infinie d'appels à `avance_Langton` ainsi qu'à une fonction `void disp_Langton(int tab[], int nbl, int nbc, int wait)` fournie qui effectue un affichage de l'état de la grille et patiente `wait` dixièmes de secondes. `wait` est indispensable pour s'assurer que votre terminal affiche correctement les images. On pourra prendre `wait=5` et ajuster cette valeur en fonction des besoins.

La fonction `disp_Langton` utilise des codes ANSI spéciaux pour un affichage dans un terminal aussi lisible que possible (et en couleurs), il est possible que votre terminal ne

les supporte pas. Vous disposez d'une fonction `void disable_color(void)` qui permet de désactiver cet affichage couleur et d'utiliser de simples symboles. L'affichage textuel est également plus rapide, désactiver la couleur peut permettre de résoudre des soucis de mise à jour trop lente.

4. Essayer la fonction `Langton` sur une grille de hauteur 15 et de largeur 45 et vérifier son bon fonctionnement.

5. Modifier la fonction `Langton` pour qu'elle n'affiche le résultat qu'une fois tous les 500 déplacements, et tester la fonction sur une grille de 125 lignes et 250 colonnes. Les dimensions du terminal seront a priori trop petites pour afficher le résultat correctement. Il est possible de réduire la taille des caractères dans le terminal pour voir la grille en entier (en général, cela se fait en maintenant la touche `CTRL` et en pressant les touches `+` et `-` du pavé numérique). Si vous disposez de Python/Matplotlib sur votre machine, vous pouvez renseigner le chemin vers `python` dans le `makefile` et utilise `make run_py` pour lancer le programme pour rediriger la sortie du terminal vers un programme Python qui l'affichera graphiquement.

Si tout fonctionne bien, après avoir eu un comportement chaotique pendant un certain temps, on voit apparaître un comportement émergent de la fourmi... on peut montrer que sur une grille infinie, ce comportement apparait toujours, quel que soient les conditions initiales de la grille si elles sont d'extension limitée.

6. Justifier qu'après un certain temps, l'écran redeviendra intégralement vierge. Vous pouvez le tester sur une grille de petite taille (3×3 par exemple, en n'oubliant pas de retirer la commande n'affichant qu'une image sur 500), mais sur une grande grille, le temps nécessaire serait bien trop long!

7. Modifier la fonction pour qu'elle gère trois fourmis (on choisira trois points de départ distincts dans la grille). On supposera que les fourmis pivotent à gauche si elles voient une valeur nulle et à droite si elles voient une valeur non-nulle, quelle qu'elle soit, et que la fourmi `i` ($i \in [\![1\,..\,3]\!]$) transforme les 0 non pas en 1 mais en i, de sorte qu'elles laissent toutes une trace de couleur différente.

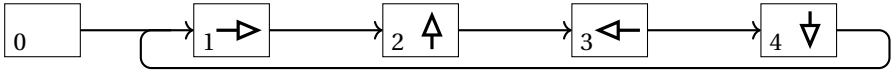
2.3 Rotorouteurs

Les rotorouteurs sont un autre type d'automate utilisant un agent, beaucoup plus récent (proposé par Jim Propp vers 2000). Ils sont liés aux problèmes, importants dans beaucoup de domaines des sciences, de « marche au hasard », tout en étant parfaitement déterministes.

On part toujours d'une grille intégralement composée de zéros. L'agent se déplaçant sur la grille n'a plus de direction propre, seulement ses coordonnés `i` et `j`. La grille en revanche peut contenir 5 états possibles, car l'agent peut déposer sur une case quelconque de la grille un « panneau » avec une flèche indiquant une direction (vers la droite, vers le haut, vers la gauche ou vers le bas). Une case contenant un panneau contiendra les valeurs 1, 2, 3 ou 4 pour chacune des directions précitées, 0 désignant une case sans panneau.

Comme la fourmi, l'agent va se « promener » sur la grille. Sa position initiale est au milieu de l'écran. À chaque itération, il obéit aux règles suivantes :

- si la case dans laquelle il se trouve est vide (c'est-à-dire contient 0), il place un panneau désignant le haut dans la case (soit 1) et retourne à sa position de départ.
- si la case n'est pas vide, il tourne le panneau d'un quart de tour dans le sens direct, puis se déplace dans la direction indiquée par la flèche.



8. Proposer une fonction appelée `avance_rotorouteur`, dont la signatureserait `avance_rotorouteur(int tab[], int nbl, int nbc, int* i, int* j)`, effectuant un déplacement de l'agent.

9. Compléter la fonction `rotorouteur(int nbl, int nbc)` de façon à ce que, sur le modèle de la fourmi de Langton, on crée un grille remplie de zéros de la taille demandée, et on exécute indéfiniment la mise à jour de l'automate suivi d'un affichage (avec la fonction `void disp_rotorouteur(int tab[], int nbl, int nbc, int wait)`.

10. Vérifier le bon fonctionnement de l'algorithme sur une grille de petite taille (9×9 par exemple). Les quatre panneaux de couleur doivent apparaître.

11. Modifier le programme pour qu'il n'affiche plus qu'une image toutes les 50000 itérations, et tester le programme avec une grille plus grande (par exemple 61×61). On pourra utiliser la sortie Python plutôt que le terminal. Quelle figure obtient-on?

Les rotorouteurs ont la propriété intéressante de visiter tout l'espace qui leur est mis à disposition. Il s'agit donc d'une façon (généralement inefficace) de trouver la sortie d'un labyrinthe, de visiter un graphe, etc. Nous aurons sans doute l'occasion d'y revenir plus tard cette année.

3 Automates cellulaires 2D

3.1 Principe

Dans cette seconde partie, nous étudierons des automates pour lesquels il n'y a plus d'agent : toutes les cases de la grille sont mises à jour simultanément conformément à des règles d'évolution, qui dépendent généralement du contenu de la case et des cases immédiatement voisines. De tels automates sont très utilisés : ils permettent la modélisation de très nombreux phénomènes (évacuation de salle, feux de forêt, avalanches, circulation automobile, épidémies...) voire d'effectuer certaines opérations, par exemple dans le cadre du traitement d'image. Il s'agit par ailleurs d'un domaine théorique très étudié en informatique.

Toute la difficulté d'implémentation est dans le « simultanément ». On ne peut pas

mettre à jour une cellule sans que cela ait des conséquences sur ses voisines, comme nous l'avons vu dans les automates de Wolfram dans le TP précédent. Nous avions alors utilisé *deux* tableaux distincts pour représenter l'état à l'instant k et celui à l'instant $k+1$. Nous allons ici utiliser une méthode quelque peu différente.

3.2 Automate de Conway

L'automate de Conway, plus connu sous le nom de « jeu de la vie », est un exemple relativement connu d'automate cellulaire à deux dimensions. Il le doit à des règles d'évolution très simples qui, pourtant, conduisent à un automate très riche.

Les cases de la grille (les « cellules ») ne peuvent se trouver que dans deux états : 0 (morte) et 1 (vivante). Une cellule vivante ne reste vivant que si, parmi ses huit voisines immédiates, il y a deux ou trois cellules vivantes. Une cellule morte revient à la vie lorsqu'il y a *exactement* trois voisines vivantes.

Initialement, les cellules sont aléatoirement dans l'un ou l'autre de ces états.

12. Proposer une fonction `int* init_Conway(int nbl, int nbc)` allouant un tableau d'entiers de taille `nbl*nbc` et le remplissant aléatoirement avec environ 20% de 1 et 80% de 0. On pourra utiliser la fonction `int rand(void)` retournant un entier positif choisi aléatoirement (entre 0 et `RANDMAX`, cette valeur étant au moins égale à 65535).

Pour des raisons de reproductibilité du code, les séquences aléatoires générées sont toujours les mêmes. Si vous voulez un état initial différent, vous pouvez utiliser la fonction `void srand(int)` qui permet de fournir un entier qui servira de « graine » au générateur de nombres aléatoires et permettra d'obtenir une séquence différente.

Il nous faut un moyen de pouvoir faire la mise à jour sans que la mise à jour d'une cellule ne perturbe ses voisines. Pour ce faire, on introduit deux états supplémentaires : 2 indiquant une cellule morte mais qui est sur le point de naître, et 3 pour une cellule vivante, mais sur le point de mourir.

13. Comment, à partir des états 0, 1, 2 et 3, savoir si on a affaire à une cellule vivante ? Comment savoir si elle sera vivante à l'étape suivante ?

14. Écrire une fonction `int compte(int tab[], int nbl, int nbc, int i, int j)` retournant le nombre de voisines vivantes.

15. En déduire une fonction `void passe_1_Conway(int tab[], int nbl, int nbc)` prend en argument un tableau ne contenant que des 0 et des 1 et mets à jour le tableau en modifiant l'état des cellules qui évolueront à l'étape suivante.

16. De même, compléter `void passe_2_Conway(int tab[], int nbl, int nbc)` qui prend en argument un tableau contenant des 0, 1, 2 et 3 et le modifie de façon à ce qu'il ne contienne plus que des 0 et des 1 en faisant évoluer convenablement les cellules dans les états 2 et 3.

17. Compléter la fonction `void Conway(int tab[], int nbl, int nbc)` de façon à ce qu'elle crée un tableau aux dimensions demandées ensemencé aléatoirement, puis indéfiniment appelle les deux fonctions précédentes suivies à chaque fois d'un appel à la fonction `void disp_Conway(int tab[], int nbl, int nbc, int wait)` qui effectuera l'affichage de l'état de la grille.

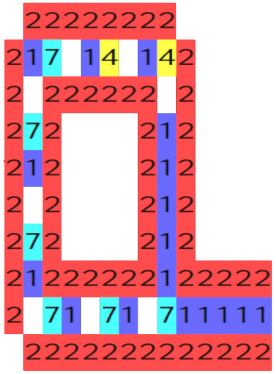
18. Essayer la fonction sur une grille de petite taille (typiquement 9×9) avec une évolution lente (une image par seconde), puis exécuter la fonction sur une grille de plus grande taille (typiquement 125×250, avec la sortie Python si elle fonctionne).

Vous pouvez, si vous le souhaitez, changer les règles d'évolution. Celles présentées ici sont celles auxquelles est parvenu John H. Conway après de nombreux essais, et c'est un des rares jeux de règles qui n'aboutissent pas rapidement soit à une extinction, soit à une explosion démographique.

En fait, cet automate est tellement riche qu'il permet de créer des portes logiques ET, OU et NON, ce qui le rend Turing-complet : n'importe quel problème algorithmique peut être écrit grâce à cet automate ! Une autre conséquence est que déterminer l'avenir de la population (extinction, stabilité, etc.) à partir de l'état initial est un problème indécidable.

3.3 Boucles de Langton

En un bel exemple d'épanadiplose, nous allons terminer cette séance de TP avec un autre automate proposé par C. Langton. La question s'est posée de savoir si des règles permettaient à une structure de se dupliquer. Puisque les automates sont une généralisation des machines de Turing, la réponse est évidemment oui [2], mais les moyens pour le faire sont intéressant. C. Langton a proposé un automate à huit états, et un motif initial, reproduit ci-dessous (les cases vides sont dans l'état 0).



Dans le motif, les 0, 1, 4 et 7 tournent dans la boucle, et lorsqu'ils arrivent à la bifurcation,

2. D'ailleurs, une variante de l'automate de Conway où une cellule à l'instant $t+1$ est vivante si et seulement si le nombre de cellules vivantes parmi ses quatre voisines immédiates est impair permet déjà de dupliquer à l'infini un motif.

se dupliquent. Lorsqu'un 7 parvient à une extrémité ouverte, il allonge le motif, lorsque c'est un 4, cela produit un coude. D'autres règles utilisant les états 3, 5 et 6 permettent de détacher une boucle complète et d'amorcer de nouvelles duplications. Ainsi, lorsque l'automate s'exécute, le motif se duplique spontanément.

Le motif est fourni sous la forme d'un tableau de type `int[10][14]` appelé `loop_motif` dans le fichier `loop.h`, déjà importé par le fichier `automates_2D.c`. Les 219 règles d'évolution se trouvent dans un tableau de type `int[219][6]` appelé `loop_rules` de ce même fichier. Il se comprend de la façon suivante : si une ligne du tableau contient les six entiers `{ 4, 0, 3, 2, 2, 1 }`, alors cela signifie que si une case est dans l'état 4 et que ses quatre voisines immédiates, *considérées dans le sens indirect*, sont respectivement dans les états 0, 3, 2 et 2, alors l'état de la cellule à l'itération suivante sera 1.

Vous êtes libres d'implémenter la simulation de la façon que vous le souhaitez, sur le modèle de ce qui a été fait précédemment. Une possibilité limitant les besoins en mémoire (et les risques de fuites) est de n'utiliser qu'un tableau, comme dans le cas de l'automate de Conway. Pour ce faire, on considérera que dans chaque case sera rangée la valeur $u + 8 \times v$ où u est l'état de la cellule à l'instant t, et v celui de la cellule à l'instant $t+1$, si celui-ci a déjà été calculé. Cela permet, à partir du reste d'une division entière par 8, d'obtenir l'état à l'instant t sans être gêné par le stockage de l'état à l'instant $t+1$, et une fois tous les nouveaux états déterminés, d'obtenir les nouveaux états en effectuant une division entière du contenu de toutes les cases par 8.

Vous disposez, comme précédemment, d'une fonction d'affichage de l'automate de signature `void disp_Loop(int tab[], int nbl, int nbc, int wait)`, mais attention, elle n'acceptera que des états entre 0 et 7 inclus.