

Propagation d'un feu de forêt

1. Principe

Nous pouvons simuler un feu de forêt qui se propage en s'appuyant sur un automate cellulaire. Un automate cellulaire fait évoluer un plan en soumettant chaque position qui le constitue à des lois de transition. Ces lois se fondent sur le voisinage des positions entre elles, nombre de voisins, qualité des voisins etc. Là il s'agit de représenter un phénomène de propagation, le feu dans une forêt.

L'algorithme original présenté ici revient à Pierre Audibert Professeur de mathématiques et d'informatique à l'Université de Paris 8.

L'espace est celui d'une matrice. Au départ il est vide et toutes les positions sont à zéro. Des arbres y sont disposés, c'est-à-dire que des positions prennent une valeur, par exemple un. Des foyers d'incendies sont ajoutés et des positions sur la valeur un des arbres prennent une valeur autre, par exemple deux. Tout est alors prêt pour la mise en œuvre du processus qui est très simple : toutes les positions de la matrice sont parcourues, si la position est un arbre et qu'il y a une position voisine sur feu, elle passe à la valeur feu ; en revanche, si la position est à feu, et bien elle passe à cendre et prend une autre valeur disons 3. A partir de ce petit moteur plusieurs extensions sont possibles. C'est souvent le cas avec des automates cellulaires. L'automate cellulaire est une figure attachante pour l'imagination parce qu'elle soumet facilement des énigmes, par exemple ici la quête d'un feu infini.

2. Mise en forme et initialisation

La solution proposée ici est minimaliste. Elle fonctionne en mode console sans affichage graphique. Comme il s'agit d'un tout petit programme, les structures de données sont globales afin d'alléger l'écriture. De ce fait la page de code ressemble à un objet avec d'une part ses variables et d'autre part les méthodes associées, pour les traitements sur ces variables.

Structures de données

Tout d'abord, la forme de l'espace est une matrice de int « P » (Plan) et sa taille est définie par deux macros. Le fonctionnement réclame une seconde matrice « NP » (Nouveau Plan) en miroir de la première afin de stocker les résultats de la transition pour l'ensemble des positions. Ce sont au début du programme les déclarations :

```
// taille du terrain
#define TX 80
#define TY 24

// les deux plans en miroir
int P[TY][TX];
int NP[TY][TX];
```

Les différentes valeurs possibles pour chaque position sont isolées avec trois macros :

```
// les différentes valeurs possibles
enum {VIDE,BOIS,FEU,CENDRE};
```

Nous avons également deux macros pour la définition d'une densité des arbres lors de l'initialisation de la forêt. Toutes ces valeurs sont modifiables facilement pour toutes sortes de tests :

```
// pour définir un seuil statistique
#define MAXI      1000
#define SEUIL     0.6*MAXI
```

L'activité du feu est contrôlée par une variable qui indique en permanence combien de case sont à FEU et signale la fin du feu lorsqu'elle vaut 0. Là encore nous avons opté pour une variable globale parce que la taille du programme le permet et qu'aucun développement ultérieur n'est prévu.

```
// réservé activité du feu
int cmpt_feu=0;
```

Initialisation des bois

L'initialisation des bois consiste à parcourir la matrice P et pour chaque position attribuer soit la valeur BOIS en fonction de la valeur de SEUIL, soit la valeur VIDE. C'est la fonction suivante :

```
void init_bois()                // 1
{
    int x,y;
    for (y=0; y<TY; y++)        // 2
        for (x=0; x<TX; x++) {
            if ((rand()%MAXI)<SEUIL) // 3
                P[y][x]=BOIS;
            else
                P[y][x]=VIDE;
        }
}
```

- (1) La fonction init_bois() ne prend pas d'argument et ne renvoie rien.
- (2) Boucles for imbriquées qui donne accès à chaque position de la matrice.
- (3) Si la valeur de retour de la fonction rand() est inférieure à la valeur de SEUIL alors la position (y, x) dans la matrice prend la valeur BOIS, sinon elle prend la valeur VIDE.

Initialisation du feu

En ce qui concerne le feu un front est allumé en bordure gauche de l'écran. C'est la fonction suivante :

```
int mise_a_feu()                // 1
{
    int y,x=1,cmpt=0;

    for ( y=0; y<TY; y++)        // 2
        if (P[y][x]==BOIS) {
            P[y][x]=FEU;
            cmpt++;
        }
    memcpy(NP,P,sizeof(int)*TY*TX); // 3

    return cmpt;                 // 4
}
```

- (1) la fonction `init_feu()` ne prend pas d'argument mais renvoie un entier.
- (2) La boucle `for` permet de parcourir une colonne. La position horizontale `x` reste toujours la même, à 1 du bord, et toutes les positions verticales sont passées en revue. Pour chacune, si elle vaut `BOIS` elle passe à `FEU` et une variable compte tous les passages à `FEU`.
- (3) La fonction standard `memcpy()` permet de recopier très rapidement un bloc mémoire dans un autre. Là nous recopions le plan `P` mis à feu dans le plan `NP` afin d'amorcer le processus de propagation (voir plus bas le processus générateur)
- (4) Pour finir, le compte du nombre de cases à feu est renvoyé à la fonction appelante.

3. Afficher

La solution proposée ici est implémentée de façon minimaliste du point de vue du graphisme. Elle fonctionne en mode console. Pour ce faire nous utilisons simplement trois fonctions : `textcolor()` pour la couleur `gotoxy()` pour le déplacement du curseur en écriture et `putchar()` pour l'affichage d'un caractère.

Seul le plan `P` fait l'objet d'un affichage. Le plan `NP` est toujours utilisé de façon transitoire pour stocker les résultat de la propagation du feu. Pour cette raison la fonction `affiche_plan()` ne prend pas de paramètre :

```
void affiche_plan()
{
    int y,x;
    for (y=0; y<TY; y++){
        for (x=0; x<TX; x++){
            switch(P[y][x]){
                case VIDE : textcolor(0); break;
                case BOIS : textcolor(6*16); break;
                case FEU : textcolor(12*16); break;
                case CENDRE : textcolor(7*16); break;
            }
            gotoxy(x,y);
            putchar(' ');
        }
    }
}
```

- (1) Le plan `P` est parcouru ligne par ligne et
- (2) selon la valeur de chaque position une couleur est sélectionnée (12 : rouge pour feu, 7 : gris pour cendre, 6 : vert olive pour bois, 0 : noir pour rien.) La couleur est multipliée par 16, ce qui revient à un décalage de 4 bits vers la gauche (<<4) afin de passer en couleur de fond et pouvoir afficher des espaces colorés.
- (3) Enfin le curseur en écriture est déplacé à la position courante et un espace est affiché dans la couleur activée par `textcolor()`.

4. Processus générateur

Fonction de propagation

Le feu se communique des cellules sur `FEU` aux cellules sur `BOIS` voisines. Les cellules sur `FEU` passent ensuite sur `CENDRE`. La fonction de propagation est la suivante :

```

void propagation() // 1
{
    int x,xo,xo,y,yn,ys; // 2

    for (y=0; y<TY; y++){ // 3
        yn=(y-1+TY)%TY; // nord
        ys=(y+1)%TY; // sud
        for (x=0; x<TX; x++){ // 4
            xo=(x-1+TX)%TX; // ouest
            xe=(x+1)%TX; // est
            if (P[y][x]==BOIS){ // si bois et // 5

                if (P[yn][x]==FEU || P[ys][x]==FEU || // feu autour
                    P[y][xo]==FEU || P[y][xe]==FEU ) {
                    NP[y][x]=FEU; // passe en feu dans miroir
                    cmpt_feu++;
                }
            }
            else if (P[y][x]==FEU){ // mais si feu
                NP[y][x]=CENDRE; // passe à cendre dans miroir
                cmpt_feu--;
            }
        }
    }
    // recopie miroir // 6
    memcpy(P, NP, sizeof(int)*TY*TX);
}

```

- (1) La fonction propagation() ne prend pas d'argument et ne renvoie rien.
- (2) Outre les variables y et x utilitaires pour passer en revue les positions de la matrice avec les boucles for imbriquées, quatre autres variables entières vont servir à désigner les positions adjacentes (est, nord, ouest, sud) à la position (y, x) courante.
- (3) Première boucle for qui passe en revue toutes les lignes de la matrice. Les positions adjacentes (y + 1) et (y - 1) sont conservées respectivement dans les variable ys (vers sud) et yn (vers nord). Notons que nous optons pour un écran circulaire, c'est-à-dire que le débordement d'un côté se retrouve de l'autre côté. Pour (y + 1) il suffit d'un modulo sur la taille de l'écran pour ramener la valeur à celle de l'autre bord de l'écran. Pour (y - 1) il faut d'abord rajouter la taille verticale TY de la matrice avant d'opérer le modulo qui ramène sur le bord opposé.
- (4) Deuxième boucle for, imbriquée dans la première, avec cette fois les positions adjacentes horizontales stockées respectivement dans xo (vers ouest) et xe (vers est). Le principe de contrôle est le même que pour la position verticale, mais cette fois c'est la largeur TX qui est utilisée.
- (5) Au centre de l'imbrication des deux boucles, le test pour connaître la valeur de la position courante (y, x) si sa valeur est BOIS un second test est effectué pour connaître si une cellule voisine est sur FEU. Si oui, la cellule de la position courante passe sur FEU à la même position dans la matrice miroir NP qui accueille les résultats. Ensuite le compteur de feux est incrémenté de un. En revanche si la position courante n'est pas sur BOIS mais sur FEU alors elle prend la valeur de CENDRE dans la matrice miroir NP et le compteur est diminué de un.
- (6) Une fois que toutes les positions ont été regardées et les valeurs modifiées sauveées dans la matrice miroir NP il faut recopier les résultats obtenus de la matrice NP vers la matrice P afin de recommencer le processus avec la nouvelle configuration. La recopie est faite avec la fonction memcpy().

Boucle du processus

La boucle du processus est dans la fonction main() :

```
int cmpt_feu=0;                                // 1

int main()
{
int fin=0;
srand(time(NULL));
while (fin!='q'){                               // 2
    if(kbhit()){                                // 3
        fin = getch();
        init_bois();
        cmpt_feu = mise_a_feu();
    }
    if(cmpt_feu){                               // 4
        propagation();
        affiche_plan();
    }
}
return 0;
}
```

- (1) La variable globale pour le comptage des feux est initialisée à 0 à sa déclaration.
- (2) La boucle d'évènements est une boucle while qui se termine avec la pression de la touche 'q' du clavier de l'ordinateur. Dans cette boucle il y a deux tests.
- (3) Si une touche quelconque est appuyée, cette touche est récupérée afin de contrôler la fin de la boucle. Mais l'objectif est de permettre des initialisations et de relancer l'action lorsqu'un feu est fini. La première fonction appelée est init_bois() pour l'initialisation de la forêt. Ensuite c'est la fonction mise_a_feu() et le nombre de feux allumés renvoyés est affecté à la variable cmpt_feu.
- (4) Si la variable cmpt_feu est différente de zéro, c'est-à-dire s'il y a des feux allumés, la propagation a lieu. Le calcul est fait une fois et la matrice P modifiée est affichée avec un appel de la fonction affiche_plan(). Lorsqu'il n'y a plus de feu, l'action s'arrête sur le dernier affichage.

5. Perspectives

De nombreux développements sont envisageables :

- ajouter du vent, différentes variétés d'arbres plus ou moins inflammables, plus ou moins serrés
- chercher un feu éternel qui renait sans cesse de ses cendres
- envisager différents types de feu qui se combattent ou s'associent entre eux
- transposer dans le domaine médical avec la simulation d'une épidémie, d'une pandémie
- éventuellement faire un simulateur de courants de pensées et de modes,...
- d'une façon générale, tout ce qui a trait à une propagation semble pouvoir être envisagé sous l'angle d'un automate cellulaire.
- passer en mode graphique avec des bibliothèques comme allegro, SDL, directX ...