PROJET 2
INFO-SUJET 5

Yan Tonglin
Deregnaucourt Lucas

Table des matières

Chapitre 1

Présentation du sujet

L'objectif de ce projet est de générer aléatoirement et récursivement des labyrinthes de tailles définies par l'utilisateur. Ces labyrinthes seront par la suite résolus, là aussi de façon récursive.

Chapitre 2

Modélisation

Le concept est assez simple : générer un labyrinthe tel qu'il existe toujours un chemin entre deux cases, puis trouver le chemin le plus court menant du point de départ au point d'arrivée choisis par l'utilisateur. Nous avons donc défini le labyrinthe comme un tableau d'entiers de taille *TAILLE* défini à 100.

Cependant, la réalisation est un peu plus complexe. En effet, nous nous pouvions pas générer le labyrinthe à l'aide d'un fichier texte où il serait déjà construit car ce dernier doit être aléatoire. De plus, la récursivité apporte une difficulté de conception.

En faisant des recherches, nous avons trouvé deux méthodes que nous avons décidé d'implémenter : l'algorithme de division récursive et l'algorithme de Eller, qui lui n'est pas récursif.

2.1 Méthode de la division récursive

L'idée générale de cette méthode est d'une simplicité enfantine : en partant d'un espace vide, on construit deux murs afin de le séparer en quatre sous-espaces puis on ouvre des cases dans ces murs afin d'obtenir des liens entre ces sous-espaces et on recommence pour chaque sous-espace ainsi formé jusqu'à ce qu'on ne puisse plus séparer aucun sous-espace en quatre.

Pour ce faire, la récursion se fait en 3 étapes :

-Construire aléatoirement deux murs pour séparer l'espace en quatre sous-espaces.

-Ouvrir des cases aléatoirement dans ces murs afin de créer des liens entre les sous-espaces.

-Recommencer pour chacun des quatre sous-espaces et ainsi de suite.

2.2 Méthode d'Eller

Ici aussi nous partons d'un espace vide, mais l'idée générale est plus sophistiquée. Le principe de cette méthode est de générer le labyrinthe ligne par

ligne.

On commencera par la première ligne en considérant que chaque case appartient à une famille différente puis, aléatoirement et de gauche à droite, on remplace certaines case par des murs tout en respectant la condition de ne pas avoir deux murs adjacents. On redéfinit ensuite les familles comme suit : une famille est composée d'une suite de cases qui ne sont pas séparées par un mur.

Une fois la première ligne obtenue, on peut passez à la deuxième. Il suffit ici de remplacer aléatoirement et de gauche à droite des cases vides par des murs tout en s'assurant qu'il y a bien au moins une case vide sous chaque famille.

Pour la troisième ligne, on copie la première, on remplace par des cases vides les murs ainsi que toutes les cases au-dessus desquelles il y a un mur. Chaque nouvelle case vide ainsi obtenue sera considérée comme appartenant à sa propre famille. On remplace ensuite aléatoirement de gauche à droite certaines cases par des murs tout en satisfaisant les conditions suivantes : deux murs ne peuvent être adjacents et si trois cases consécutives sont de la même famille, celle du milieu sera remplacée par un mur.

Une fois cette troisième ligne complétée, on répète les opérations effectuées pour la deuxième et la troisième ligne. On dira alors que deux cases sont de la même famille s'il existe une chemin qui les relit. Pour la dernière ligne, une fois complétée, on supprime de gauche à droite les murs qui séparent deux cases qui ne sont pas de la même famille.

Au vu de cette construction, il nous a semblé nécessaire de séparer la première ligne, la dernière ligne, les lignes paires ainsi que les lignes impaires.

Au final, nous obtenons un labyrinthe où toutes les cases appartiennent à la même famille.

2.3 Résolution

La résolution a été très simple à modéliser : en effet il suffit, à partir de la case de départ, de vérifier si les cases adjacentes sont des chemins, des murs, ou l'arrivée.

La récursion consiste finalement à recommencer ces vérifications en partant des cases adjacentes à la case de départ, puis des cases adjacentes aux cases adjacentes de la case de départ et ainsi de suite.

Chapitre 3

Programmation

Nous avions l'intention d'implémenter un affichage graphique en utilisant la bibliothèque SDL, mais au vu du peu de temps qu'il nous restait après les examens, nous nous sommes retranchés sur un affichage terminal. Nous avons aussi choisi de beaucoup segmenter le code afin de le rendre le plus clair et le plus modulable possible.

3.1 Fonctions communes

```
laby initialisation(int n)
{
        int i, j;
        laby l;
        l.dim = n;
        for (i=1;i<l.dim-1;i++)
                for (j=1;j<l.dim-1;j++)
                        l.c[i][j] = 0;
        for (i=0;i<l.dim;i++)
        {
                l.c[0][i] = MUR;
                l.c[l.dim-1][i] = MUR;
                l.c[i][0] = MUR;
                l.c[i][l.dim-1] = MUR;
        }
        return l;
}
```

Comme son nom l'indique, cette fonction initialise le labyrinthe. Autrement dit, elle le vide et l'encadre par des murs. Celà nous permet d'être sûrs de travailler dans un espace « propre » par la suite.

```
void affichage1(laby l)
{
        int i,j;
        printf("\n");
        for (i=0;i<l.dim;i++)
        {
                for (j=0;j<l.dim;j++)
                {
                        if (l.c[i][j] == MUR) printf("# ");
                        else printf("  ");
                }
                printf("\n");
        }
}
```

Cette fonction permet le premier affichage du labyrinthe. En effet, avant de demander à l'utilisateur quel labyrinthe il veut résoudre, il est nécessaire de les afficher.

```
void affichage(laby l)
{
        int i,j;
        printf("\n");
        for (i=0;i<l.dim;i++)
        {
                for (j=0;j<l.dim;j++)
                {
                        if (l.c[i][j] == MUR) printf("# ");
                        else        if (l.c[i][j] == 1)
                                                printf("O ");
                                    else printf("  ");
                }
                printf("\n");
        }
}
```

Cette fonction nous permet d'afficher le labyrinthe une fois résolu. La solution sera représenté par un chemin de « O ».

3.2 Division récursive

```
int divi(int borneH,int borneG,int borneB,int borneD)
/*vérifier si division est possible: 0 pour impossible, 1 pour
↪  possible*/
{
        int res;
        if (((borneB-borneH) == 2)||((borneD-borneG) == 2))
                res = 0;
        else res = 1;
        return res;
}
```

La fonction *divi* nous permet simplement de vérifier si le sous-espace est divisible ou non. Ce sera la condition de sortie de notre récursion.

```
laby division(laby l,int borneH,int borneG,int borneB,int borneD)
{
        int booleen;
        int lig,col,i;
        int h,b,g,d;
        int ferme;
        booleen = divi(borneH,borneG,borneB,borneD);
        if (booleen == 1)
        {
                lig = 2*(rand()%((borneB-borneH-1)/2)+1)+borneH;
                col = 2*(rand()%((borneD-borneG-1)/2)+1)+borneG;
                h = 2*(rand()%((lig-borneH)/2))+1+borneH;
                b = 2*(rand()%((borneB-lig)/2))+lig+1;
                g = 2*(rand()%((col-borneG)/2))+1+borneG;
                d = 2*(rand()%((borneD-col)/2))+col+1;
                ferme = rand()%4;
                switch (ferme)
                {
                        case 0:h = TAILLE-1;break;
                        case 1:b = TAILLE-1;break;
                        case 2:g = TAILLE-1;break;
                        case 3:d = TAILLE-1;break;
                }
                for (i=borneH+1;i<borneB;i++)
                        if (i != h && i != b)
                                l.c[i][col] = MUR;
                for (i=borneG+1;i<borneD;i++)
                        if (i != g && i != d)
                                l.c[lig][i] = MUR;
```

```
                l = division(l,borneH,borneG,lig,col);
                l = division(l,lig,borneG,borneB,col);
                l = division(l,borneH,col,lig,borneD);
                l = division(l,lig,col,borneB,borneD);
        }
        return l;
}
```

Comme dit précédemment, la condition de sortie de cette fonction récursive est la possiblilité de diviser par quatre le sous-espace ou non. Tout d'abord, on obtient deux lignes aléatoires, une horizontale et une verticale. Elles permettent de séparer le labyrinthe en quatre sous-espaces. Ensuite, on choisit aléatoirement une direction telle que l'on garde cette direction fermée et on ouvre une case dans chacune des trois autres directions afin de former un chemin entre les sous-espaces. Enfin, on accède au sous-espace en haut à gauche et on itère cette méthode jusqu'à il ne soit plus divisible.

3.3 Algorithme d'Eller

```
laby preLigne(laby l,int lig)
{
        int col,dep = 1,addmur;
        int i,set = 1;
        dep = 1;
        for (i=2;i<l.dim-1;i+=2)
        {
                addmur = rand()%2;
                if (addmur == 1)
                {
                        for (col=dep;col<i;col++)
                                l.c[lig][col]=set;
                        l.c[lig][i] = MUR;
                        set++;
                        dep = i+1;
                }
        }
        for (col=dep;col<i;col++)
                l.c[lig][col] = set;
        return l;
}
```

Cette fonction nous permet d'obtenir la première ligne du labyrinthe telle que nous l'avons décrite dans la partie *modélisation*. L'aléatoire est ici représenté par la variable *addmur* et les différentes familles par la valeur de la variable *set*.

```
laby lignePair(laby l,int lig)
{
        int i,col,nbblc,w,t,arv,dep=1,set=1,blc[l.dim];
        for (i=1;i<l.dim-1;i++)
                l.c[lig][i]=MUR;
        while (dep < l.dim-1)
                 {
                arv = dep+2;
                while (arv < l.dim-1 && l.c[lig-1][arv] ==
                ↪  l.c[lig-1][dep])
                        arv += 2;
                nbblc = rand()%((arv-dep)/2)+1;
                {
                        for (i=0;i<((arv-dep)/2);i++) /*ligne
                        ↪  12*/
                                        blc[i] = i+1;
                        for (i=0;i<nbblc;i++)
```

```
                        {
                                w = rand()%((arv-dep)/2-i)+i;
                                t = blc[i];
                                blc[i] = blc[w];
                                blc[w] = t; /*ligne 19*/
                                col = 2*(blc[i]-1)+dep;
                                l.c[lig][col] = l.c[lig-1][col];
                        }
                }
                dep = arv;
        }
        return l;
}
```

Cette fonction sert à compléter les lignes paires. La ligne est tout d'abord remplie avec des murs puis, aléatoirement, au moins une ouverture est créée sous chaque famille. Les lignes 12 à 19 servent à obtenir nbblc nombres aléatoires différents entre 1 et $\frac{dep-arv}{2}$.

La fonction *ligneImpair* est trop conséquente pour qu'il soit pertinent de la mettre dans le rapport. Elle suit simplement la méthode de construction des lignes impaires vues précédemment.

```
laby derLigne(laby l,int lig)
{
        int i,j,col;
        for (i=1;i<l.dim-1;i+=2)
                if (l.c[lig-1][i] != MUR)
                        l.c[lig][i] = l.c[lig-1][i];
        for (i=1;i<l.dim-2;i+=2)
        {
                j = i+2;
                while ((l.c[lig][i]!=l.c[lig][j] ||
                ↪  (l.c[lig][i]==0) && j<l.dim-1)
                        j += 2;
                if (l.c[lig][i] == l.c[lig][j] && l.c[lig][i] !=
                ↪  0)
                {
                        col = 2*(rand()%((j-i)/2))+i+1;
                        l.c[lig][col] = MUR;
                }
        }
        return l;
}
```

Cette fonction vérifie tout d'abord pour chaque case de la dernière ligne si la case au-dessus est un mur ou non. Si ce n'est pas un mur, alors la case ap-

partient à la même famille que la case au-dessus d'elle. On supprime ensuite les murs qui séparent deux cases appartenant à la même famille.

```
laby eller(laby l)
{
        int i,j;
        l = preLigne(l,1);
        for (i=2;i<l.dim-2;i++)
        {
                if (i%2 == 0)
                        l = lignePair(l,i);
                else
                        l = ligneImpair(l,i);
        }
        l = derLigne(l,l.dim-2);
        for (i=0;i<l.dim-1;i++)
                for (j=0;j<l.dim-1;j++)
                        if (l.c[i][j] != MUR)
                                l.c[i][j] = 0;
        return l;
}
```

Cette fonction « principale » utilise toutes les précédentes. Elle permet de générer le labyrinthe ligne par ligne de façon itérative. A la fin de cette fonction, on fait appartenir à la même famille toutes les cases qui ne sont pas des murs.

3.4 Résolution

```
int direction(laby l, int lig, int col)
{
        int t[4] =
        ↪  {l.c[lig+1][col],l.c[lig][col+1],l.c[lig-1][col],l.c[lig][col-1]};
        int indice = 0,i;
        for (i=1;i<4;i++)
                if (t[i] < t[indice]) indice=i;
        return indice;
}
```

Cette fonction renvoie la direction dans laquelle l'algorithme va se déplacer lors de la récursion. Le tableau t contient les cases adjacentes (pas diagonalement) à la case [lig][col] du labyrinthe. Elle renvoie 1 pour bas, 2 pour droite, 3 pour haut, 4 pour gauche.

```
int auFond(laby l,int lig,int col,int indice)
{
        int
        ↪  t[4]={l.c[lig+1][col],l.c[lig][col+1],l.c[lig-1][col],l.c[lig][col-1]};
        int somme = 0,i,res;
                for (i=0;i<4;i++)
                        if (i!=indice) somme += t[i];
        if (somme == 30) return 1;
        else return 0;
}
```

Cette fonction renvoie 1 si l'on a atteint un cul-de-sac et 0 sinon.

```
int estArrive(int x1,int y1,int x2,int y2)
{
        if (x1 == x2 && y1 == y2) return 1;
        else return 0;
}
```

L'objectif de cette fonction est on ne peut plus clair : renvoyer 1 si l'on a atteint l'arrivée et 0 sinon.

```
laby resolve(laby l,int lig,int col,int x2,int y2)
{
        int direc;
        if (estArrive(lig,col,x2,y2) == 0)
        {
                direc = direction(l,lig,col);
                if (auFond(l,lig,col,direc) == 1)
                        l.c[lig][col] += 2;
                else l.c[lig][col] += 1;
                switch (direc)
                {
                        case 0: l = resolve(l,lig+1,col,x2,y2);
                        ↪  break;
                        case 1: l = resolve(l,lig,col+1,x2,y2);
                        ↪  break;
                        case 2: l = resolve(l,lig-1,col,x2,y2);
                        ↪  break;
                        case 3: l = resolve(l,lig,col-1,x2,y2);
                        ↪  break;
                }
        }
        return l;
}
```

La condition d'arrêt est évidemment le « booléen » *estArrive*. S'il est faux, on « marque » la case de coordonnées [lig][col] puis on effectue la récursion sur la case en-dessous, la case à droite, la case au-dessus et enfin la case à gauche. Les fonctions *estChemin* et *chemin* permettent ensuite de détecter les cases qui constituent la solution et de les homogénéiser en leur donnant la valeur 1 et en donnant 0 aux cases qui ne sont ni un mur, ni une partie de la solution.

3.5 Jeux d'essais

Nous allons essayer notre programme avec une dimension n=15 et, pour la résolution, en partant en haut à gauche afin d'arriver en bas à droite du labyrinthe généré par la méthode de la division récursive.

FIGURE 3.1 –

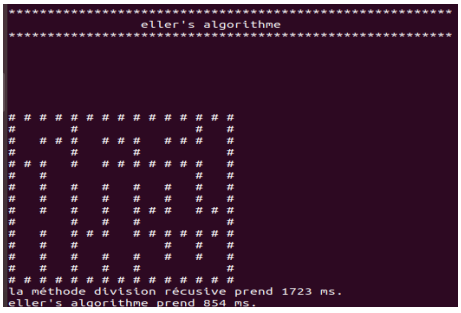


FIGURE 3.2 –

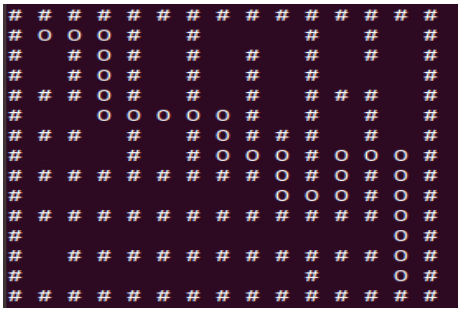


FIGURE 3.3 –

Essayons maintenant pour n = 25 :

FIGURE 3.4 –

FIGURE 3.5 –

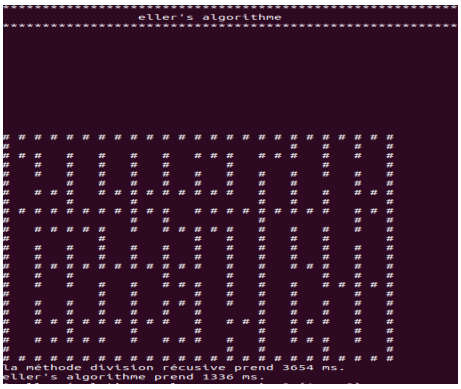
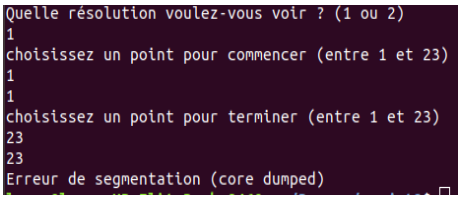


FIGURE 3.6 –



Nous avons remarqué qu'à partir d'une certaine dimension, la résolution ne marche plus et le programme renvoie un message d'erreur. Nous pouvons cependant voir que les labyrinthes ont tous une forme singulière, ce qui montre bien qu'ils sont générés aléatoirement et sans biais.

Chapitre 4

Conclusion

L'implémentation de la méthode d'Eller nous a permis de comparer une méthode itérative et une méthode récursive de génération de labyrinthe aléatoire. Les observations sont contre-intuitives : l'algorithme d'Eller, donc itératif, est plus rapide que l'algorithme itératif. Celà vient certainement du fait que la méthode de division récursive n'est pas l'une des plus optimisées, contrairement à la méthode d'Eller.

En ce qui concerne la résolution, elle ne fonctionne que si le chemin n'est pas trop long et que la dimension du labyrinthe est inférieure ou égale à 17. Au-delà de ces limites, le programme renvoie un *« segmentation fault (core dumped) »*. Au début nous pensions qu'il s'agissait d'un dépassement de tableau, mais en utilisant un debugger, nous nous sommes aperçus que la résolution s'arrêtait en plein milieu du labyrinthe. Nous en avons donc déduit qu'il s'agit du dépassement de pile. Ce problème est dû au coût assez élevé de la récursion, ce qui doit vite saturer la mémoire et provoque donc l'arrêt du programme.