1 Création d'un labyrinthe

L'algorithme employé pour la génération de labyrinthes utilise des propriétés qui s'apparentent au *Quick Union Find*; algorithme de recherche et de construction de graphes de connexité dans un ensemble d'éléments donné.
Les labyrinthes créés ici sont au minimum 1-connexes [1] ; à partir de ce type de labyrinthe [2], nous pouvons à tout moment augmenter la connexité en cassant des murs, et de ce fait créer d'autres chemins possibles [3].
L'embryon du labyrinthe est tel que toutes les positions de départ (les nœuds) sont entourées de 4 murs. Il se présente sous la forme d'une matrice d'entiers de dimensions ($N \leftarrow 2n+1, M \leftarrow 2m+1$) où $n \times m$ est le nombre de cases non-murées. Dans chacune de ces cases, un identifiant unique est placé (de 0 à $n \times m - 1$). Nous posons la valeur -1 pour les cases murées. À l'initialisation, nous obtenons la matrice L pour $n \leftarrow m \leftarrow 3$:

$$L \leftarrow \begin{pmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & \mathbf{0} & -1 & \mathbf{1} & -1 & \mathbf{2} & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & \mathbf{3} & -1 & \mathbf{4} & -1 & \mathbf{5} & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & \mathbf{6} & -1 & \mathbf{7} & -1 & \mathbf{8} & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}$$

Chacune des cases non-murées se trouve dans une des composantes d'un même graphe ; chaque composante est réduite à un seul nœud. Pour $n = m = 3$, un graphe à 9 composantes connexes est créé.

Afin d'obtenir un graphe à une composante connexe, nous sélectionnons aléatoirement un mur séparant deux composantes disjointes ; le mur est détruit et l'identifiant d'une des deux composantes est propagé sur les nœuds de l'autre. Nous choisissons de propager la valeur du plus petit identifiant.

Quand les $n \times m$ nœuds de départ prennent la valeur [4] 0 (il y en a déjà une qui a la valeur 0 à l'initialisation), alors toutes les composantes du graphe sont jointes ; l'unique graphe résultant est 1-connexe [5].

1.1 Détail de l'algorithme

1. Initialisation :
   - Une matrice $L[N \leftarrow 2n+1][M \leftarrow 2m+1]$ donnée (allouée) ;
   - $k \leftarrow 0$;
   - Pour i de 0 à $N-1$ (toutes les lignes) faire :
     - Pour j de 0 à $M-1$ (toutes les colonnes) faire :
       - Si (i impair et j impair) alors :
         - $L[i][j] \leftarrow k$;
         - $k \leftarrow k+1$;
       - Sinon : $L[i][j] \leftarrow -1$;
       - FIN "Si" ;
     - FIN "Pour" ;
   - FIN "Pour" ;

1. Pour tout point (emplacement non muré) dans ce labyrinthe, il existe un et un seul chemin le reliant à n'importe quel autre point.
2. Assimilable à un graphe acyclique.
3. Assimilable à un graphe cyclique.
4. Dans le cadre d'une représentation matricielle, pour un nœud du graphe, la valeur contenue dans une case à l'initialisation doit être différente de -1.
5. Graphe à une composante 1-connexe.

2. Pour $NbCasesAZero$, le nombre de cases à 0 : $NbCasesAZero \leftarrow 1$;
3. Tant que $NbCasesAZero < n \times m$ faire :
   — Prendre au hasard (x, y) tel que :
   $L[y][x] = -1$ et (x impair ou EXLUSIF y impair [6]) ;
   — Si x impair (mur qui sépare Nord-Sud) alors :
      — $d \leftarrow L[y-1][x] - L[y+1][x]$;
      — Si $d > 0$ alors :
         — $L[y][x] \leftarrow L[y+1][x]$;
         — Propager la valeur $L[y+1][x]$ à partir du point $(x, y-1)$ (en 4-connexité) pour les cases contenant la valeur $L[y-1][x]$ [7] ;
      — Sinon, si $d < 0$ alors :
         — $L[y][x] \leftarrow L[y-1][x]$;
         — Propager la valeur $L[y-1][x]$ à partir du point $(x, y+1)$ (en 4-connexité) pour les cases contenant la valeur $L[y+1][x]$ [7] ;
   — Sinon (y impair, mur qui sépare Est-Ouest) :
      — $d \leftarrow L[y][x-1] - L[y][x+1]$;
      — Si $d > 0$ alors :
         — $L[y][x] \leftarrow L[y][x+1]$;
         — Propager la valeur $L[y][x+1]$ à partir du point $(x-1, y)$ (en 4-connexité) pour les cases contenant la valeur $L[y][x-1]$ [7] ;
      — Sinon, si $d < 0$ alors :
         — $L[y][x] \leftarrow L[y][x-1]$;
         — Propager la valeur $L[y][x-1]$ à partir du point $(x+1, y)$ (en 4-connexité) pour les cases contenant la valeur $L[y][x+1]$ [7] ;
4. FIN “Tant que”.

1.2 Exemple de propagation et résultat

+	+	+	+	+	+	+		+	+	+	+	+	+	+
+	0	+	**1**	**1**	**1**	+		+	0	+	**0**	0	**0**	+
+	0	+	+	+	**1**	+		+	0	+	+	+	0	+
+	0	0	0	**+**	**1**	+	→	+	0	0	0	0	**0**	+
+	+	0	+	+	+	+		+	+	0	+	+	+	+
+	0	0	0	0	0	+		+	0	0	0	0	0	+
+	+	+	+	+	+	+		+	+	+	+	+	+	+

$$NbCasesAZero \leftarrow NbCasesAZero + 3$$

Nous obtenons pour $N = M = 200$ (matrice de 401×401) le labyrinthe figure 1 :

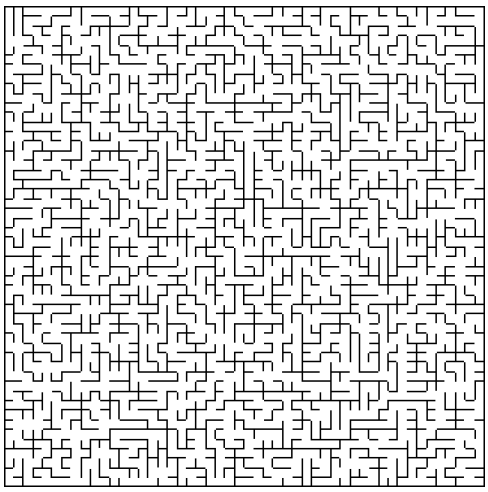


FIGURE 1 – Labyrinthe ($N = M = 200$) généré aléatoirement.

6. Si x et y sont tous les deux impairs, alors $L[y][x] \neq -1$
7. Pendant la propagation, incrémenter $NbCasesAZero$ s'il y a lieu (comme décrit plus haut).

2 Le plus court chemin dans un labyrinthe

Nous allons décrire ici un algorithme pour calculer les plus courts chemins dans un labyrinthe ; cette méthode est basée sur l'algorithme du même nom introduit par E.W. Dijkstra en 1959. Il s'agit de trouver le plus court chemin dans un graphe orienté $G = (S, A)$. Dans ce graphe, les sommets sont reliés entre eux par des arêtes, chacune est notée a_i et possède un poids p_i. Le coût de parcours du chemin $C = \{a_1, \cdots, a_n\}$ équivaut à la somme des poids des arêtes qui le constituent. Dans le cas particulier du labyrinthe [8], tous les p_i valent 1. Nous utilisons la matrice du labyrinthe pour stocker les informations liées au coût de déplacement, cf. figure 3.

2.1 Algorithme simple, récursif (pas encore celui de Dijkstra qui nécessite une file)

$Pcc(x_1, y_1, x_2, y_2, v)$
Début :
$v \leftarrow v + 1$;
$L[y_1][x_1] \leftarrow v$;
Si $x_1 = x_2$ et $y_1 = y_2$ alors :
retour ;
Pour chaque $d(x_{dir}, y_{dir})$ pris parmi les quatre directions possibles :
Si $v < L[y_1 + y_{dir}][x_1 + x_{dir}]$ ou $L[y_1 + y_{dir}][x_1 + x_{dir}] = 0$ alors :
$Pcc(x_1 + x_{dir}, y_1 + y_{dir}, x_2, y_2, v)$;
Fin.

1. Initialiser le labyrinthe [9] $M \times N$;
2. Pour un point de départ (x_d, y_d) et un point d'arrivée (x_f, y_f) faire : $Pcc(x_d, y_d, x_f, y_f, 0)$; cela va remplir, dans un parcours en profondeur d'abord, l'ensemble des positions non-murées du labyrinthe en marquant la distance la plus courte depuis le point de départ. Ici (voir Figure 2), nous ne pouvons arrêter l'exploration dès la rencontre du point d'arrivée, car le premier chemin trouvé n'est pas nécessairement le plus court. La priorité est celle de l'ordre dans lequel on explore le voisinage d'une position, cette priorité ne considère donc pas la distance.
3. Partir du point d'arrivée et reconstruire le chemin (pour un point à valeur v, prendre le voisin qui a pour valeur $v - 1$) jusqu'au point de départ ;

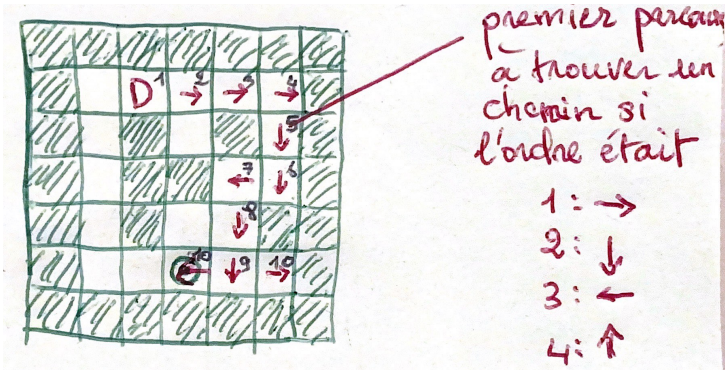


FIGURE 2 – Exemple d'un premier parcours aboutissant au point d'arrivée mais qui n'est pas « le plus court chemin ».

8. Ici un graphe non pondéré
9. Nous obtenons une matrice L / $\forall a_{i,j} \in L$; $a_{i,j} \in \{-1, 0\}$.

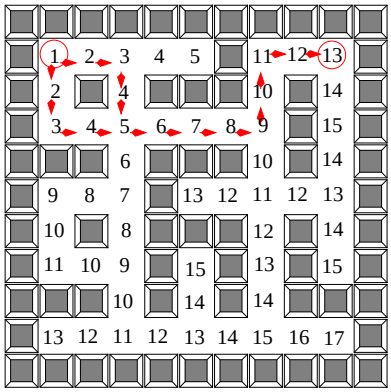


FIGURE 3 – Schéma de résolution du plus court chemin dans un labyrinthe, après un parcours (en profondeur d'abord) complet des possibles.

Voici un exemple d'exécution du programme :

```
bash$ ./labyrinthe
Entrer les dimensions du labyrinthe :
-> Largeur : 3
-> Hauteur : 3

****************************
* Generation du labyrinthe *
****************************

+ + + + + + +
+   +       +
+   + + +   +
+       +   +
+ +   + + + +
+           +
+ + + + + + +

Entrer les coordonnees du point de depat :
-> X : 1
-> Y : 1

+ + + + + + +
+ D +       +
+   + + +   +
+       +   +
+ +   + + + +
+           +
+ + + + + + +

Entrer les coordonnees du point d'arrivee :
-> X : 5
-> Y : 1

+ + + + + + +
+ D +       +
+   + + +   +
+       +   +
+ +   + + + +
+ A         +
+ + + + + + +

********************************
* Calcul du plus court chemin *
********************************

+ + + + + + +
+ D +       +
+ . + + +   +
+ . .   +   +
+ + . + + + +
+ A .       +
+ + + + + + +

Recommencer (o/n) ?
-> n

*******
* Fin *
*******

bash$
```

2.2 Et pour faire un Dijkstra ...

L'idée du Dijkstra est simple, on explore autour du point de départ en allant – **de partout** – de plus en plus loin (un peu comme si on traçait des cercles dont le rayon est de plus en plus grand). Dans un arbre explorant les possibles à partir d'un nœud, cela revient à privilégier le parcours en largeur, soit on explore tout l'étage d'en dessous avant d'aller au suivant.

Ce mode d'exploration nécessite donc de mémoriser pour prioriser les positions à proximité sur les positions plus éloignées, en disant : " le premier arrivé est le premier servi " le tout dans une file d'attente. Ceci peut donc être réalisé à l'aide d'une file (strucutre FIFO).

La file dont nous aurons besoin doit permettre de mémoriser une position en l'**enfilant (enfile)** à la fin de la file, de **restituer (défile)** la première position de la file et d'indiquer si elle **est vide**.

La Figure 4 donne un aperçu des premières étapes d'un parcours en largeur utilisant une file.

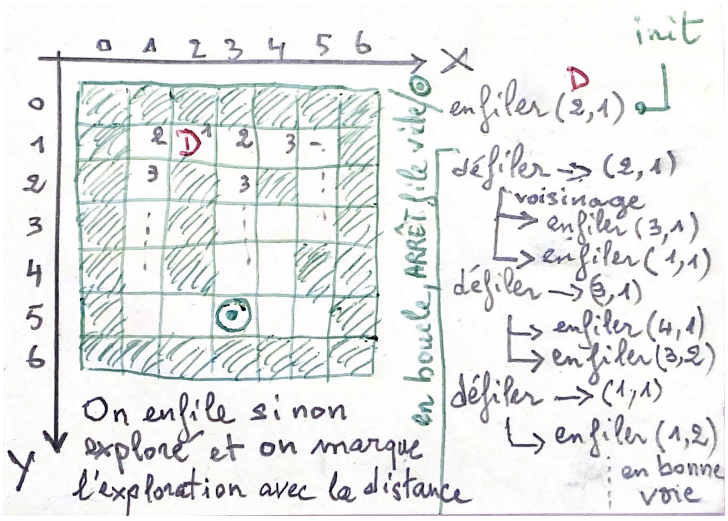


FIGURE 4 – Étapes principales du démarrage d'une exploration en largeur (autour du point de départ).

2.3 Grandes lignes de l'algorithme

1. Pour un labyrithe L initialisé avec des 0 (vide) et des -1 (mur) ;
2. Pour $D \leftarrow (x_d, y_d)$ et $A \leftarrow (x_a, y_a)$ respectivement la position de départ et celle d'arrivée ;
3. Avoir une $file$ vide capable de stocker des positions (x, y) ;
4. $enfiler(D)$;
5. Mettre d, la distance, à 1 ;
6. $L[y_d][x_d] \leftarrow d$;
7. Tant que $\neg est\_vide()$:
   - $d \leftarrow d + 1$;
   - $p \leftarrow defiler()$
   - Pour tous les $v \leftarrow (x_v, y_v)$ un des voisins 4-connexe de p :
     - Si $L[y_v][x_v] = 0$:
       - $L[y_v][x_v] = d$;
       - Si $A = v$ alors vider la file et **sortir de la boucle**.
       - Sinon $enfiler(v)$
     - Fin "Si" ;
   - FIN "Pour" ;
8. FIN "Tant que".
9. Partir du point d'arrivée et reconstruire le chemin (pour un point à valeur v, prendre le voisin qui a pour valeur $v - 1$) jusqu'au point de départ ;

2.4 Un Dijkstra sans file ... et avec un automate cellulaire

Pour cette dernière partie, nous vous invitons à reprendre le support portant sur les automates cellulaires (le feu de forêt). Il nous servira à créer une autre méthode de résolution du plus court chemin.

2.4.1 Générer et afficher le labyrinthe dans une application *GL4dummies*

On va commencer par reprendre l'algorithme de génération de labyrinthes, le principal est de produire une carte (map / image) à la résolution du *screen GL4Dummies* (ou inversement, c'est-à-dire faire en sorte que le *screen* soit aux dimensions choisies pour le labyrinthe).

Concernant les valeurs, un `int` à -1, interprété en non signé est égal à `0xFFFFFFFF`, soit blanc, le zéro est `0x0`, et donne bien du noir. On garde donc ces valeurs, et en générant le labyrinthe dans la mémoire du `screen`[10], nous obtenons un équivalent au résultat montré en Figure 5.

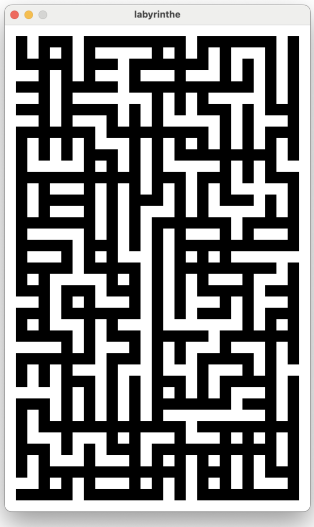


FIGURE 5 – Labyrinthe généré directement dans le *screen*.

On propose de modifier le début du code obtenu à l'issu de la réalisation de la version simple[11] du *feu de forêt*.

```
/* la largeur (nombre de colonnes de la matrice) */
static const int _m = 13, _M = (_m << 1) + 1;
/* la hauteur (nombre de lignes de la matrice) */
static const int _n = 21, _N = (_n << 1) + 1;

enum ca_states_t {
  WALL = -1,
  EMPTY = 0,
  EXPLORED = RGBA(0, 255, 0, 0), /* important (voir _is_explored)
                                    rouge à zéro et vert à 255 */
  PATH = RGBA(255, 0, 0, 0)  /* important (voir _is_path)
                                    rouge à 255 et vert à zéro */
};
```

puis on modifie le début du `main` afin coller aux dimensions du labyrinthe et appeler la bonne fonction d'initialisation.

10. Il suffit de *caster* le pointeur vers les pixels en `GLint *` et le tour est joué.
11. Code obtenu en suivant les consignes jusqu'à l'avant dernière page, soit sans l'exercice proposé.

```
int main(int argc, char ** argv) {
  if(!gl4duwCreateWindow(argc, argv, /* args du programme */
                         "labyrinthe", 0, 0, /* titre et (x, y) de la fenêtre */
                         _M << 4, _N << 4, /* largeur (16 * M), hauteur (16 *
                              N) */
                         GL4DW_SHOWN) /* état visible */) {
    return 1;/* code d'erreur - echec d'ouverture de fenêtre */
  }
  /* on force la synchronisation verticale */
  SDL_GL_SetSwapInterval(1);

  gl4dpInitScreenWithDimensions(_M, _N);
  /* toujours le même rand pour les tests */
  srand(42);
  /* ou un rand qui change : srand(time(NULL)); */
  init_maze(); /* à vous de la prototyper et de la définir (écrire) */
  gl4duwIdleFunc(cellular_automaton_step);

  /* le reste est comme d'habitude ... */
  atexit(ciao);
  gl4duwDisplayFunc(dessine);
  gl4duwMainLoop();
  return 0;
}
```

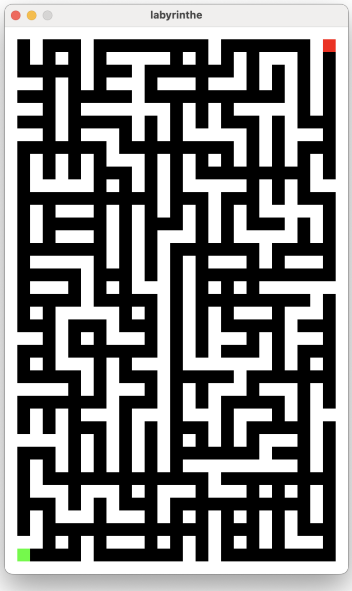
À vous de reprendre les explications et le pseudo-code de la section 1 afin d'écrire la fonction `init_maze`. Vous pourrez utiliser les constantes `EMPTY` et `WALL` respectivement à la place de 0 et -1 afin d'indiquer une position non murée et une position murée. Aussi, pensez à commenter une bonne partie de la fonction `cellular_automaton_step` car les constantes ont changé et qu'il va falloir revoir le fonctionnement de l'automate à la section suivante.

2.4.2 Chercher le chemin vers l'arrivée

On propose une réinterprétation de l'algorithme de Dijkstra sans utiliser une file. Commençons par écrire la remplaçante de `light_the_fire` (à appeler juste après `init_maze`), elle se charge juste de placer le point de départ et le point d'arrivée. Nous proposons d'utiliser la constante `EXPLORED` comme valeur du pixel au point de départ et `PATH` comme valeur du pixel au point d'arrivée. Cette fonction pourrait être :

```
void init_positions(void) {
  GLuint w = gl4dpGetWidth();
  GLuint h = gl4dpGetHeight();
  GLuint * lab = gl4dpGetPixels();
  /* en bas à gauche */
  lab[1 * w + 1] = EXPLORED;
  /* en haut à droite */
  lab[(h - 2) * w + (w - 2)] = PATH;
}
```

où on aurait mis le point de départ en bas à gauche et le point d'arrivée en haut à droite. Ce qui donne visuellement :



Avant d'écrire la nouvelle version de l'automate, nous allons avoir besoin de plusieurs fonctions qui encodent/décodent le coût de l'exploration et du chemin dans la couleur tout en gardant le même aspect de cette dernière. Comme les constantes `EXPLORED` et `PATH` utilisent chacune une composante distincte de couleur (respectivement le vert et le rouge), nous choisissons d'utiliser ce qui reste pour encoder un coût sans que cela ne se voit "trop".

Déjà la composante alpha n'est pas utilisée dans cet example, on peut donc entièrement s'en servire pour mettre le coût. Par contre nous seront limités à des valeurs allant jusqu'à 255 (car un seul octet). On va donc ajouter 6 bits de la composante bleue ce qui nous permet de multiplier par 63 (ou `0x3F`), car $2^6 - 1 = 63$, la valeur maximale possible pour un coût, ce qui nous donne : $256 \times 63 + 255 = 16383$. Cela est plutôt suffisent pour un chemin dans une image haute définition.

Le choix de n'utiliser que 6 bits du bleu fait que nous ne percevrons pas trop de changement dans la teinte, on gardera ainsi des couleurs cohérentes entre positions explorées et positions faisant partie du meilleur chemin.

Voici un exemple de fonctions qui vont nous permettre de réaliser cette manipulation (vous pouvez les mettre dans `tiny4D.h`) :

```
static inline int _is_explored(GLuint color) {
  GLubyte r = _red(color);
  GLubyte g = _green(color);
  GLubyte b = _blue(color);
  return (r == 0 && g == 255 && b <= 0x3F /* 63 */);
}

static inline int _is_path(GLuint color) {
  GLubyte r = _red(color);
  GLubyte g = _green(color);
  GLubyte b = _blue(color);
  return (r == 255 && g == 0 && b <= 0x3F /* 63 */);
}

static inline int _get_cost(GLuint color) {
  if(_is_path(color) || _is_explored(color)) {
    GLubyte b = _blue(color);
    GLubyte a = _alpha(color);
    return b * 256 + a;
  }
  return 0;
}
```

```
static inline GLuint _make_cost(GLuint base_color, int cost) {
  return _rgba(_red(base_color), _green(base_color), (cost >> 8) & 0x3F, cost &
      0xFF);
}
```

Tout est ainsi prêt pour proposer un automate d'exploration des chemins depuis le point de départ. Les règles sont ici plus simples que celles du feu de forêt. Il s'agit de traiter uniquement les cases `EMPTY` et les cases `PATH` (notez que la seule case `PATH` au début est la position d'arrivée). Chaque fois qu'une de ces cases est en contact (4-connexité) avec une case `EXPLORED`, elle propage le coût de cette dernière en ajoutant 1. Si la case est `EMPTY`, elle devient `EXPLORED`, si elle est `PATH` alors l'exploration est TERMINÉE.

Pour différencier la phase d'exploration de la phase de dessin du plus court chemin, nous proposons de modifier la *idle func* quand le point d'arrivée est atteint (fin de l'exploration). Il faudra donc une seconde fonction, exemple *next cell automaton step*, pour finir le travail dans l'autre sens : revenir depuis le point de d'arrivée vers le point de départ en coloriant en rouge la case *de type explored*, en contact avec une case *de type path*. La case voisine *de type path* (en (nx, ny)) doit avoir un coût strictement égal à celui de l'actuelle case (en (x, y)) **plus un** (utilisez `_get_cost` et `_make_cost`). **Nous vous laissons finir cette dernière partie coloriant *le plus court chemin trouvé*** ; un DM sera ajouté au moodle pour pouvoir déposer votre travail.

Voici donc une version de l'automate cellulaire de la première partie, celle réalisant l'exploration :

```
void cellular_automaton_step(void) {
  GLuint i;
  GLuint w = gl4dpGetWidth();
  GLuint h = gl4dpGetHeight();
  GLuint wh = w * h;
  GLuint * lab = gl4dpGetPixels();
  GLuint * lab_copy = malloc( wh * sizeof *lab_copy ); assert(lab_copy);
  memcpy(lab_copy , lab , wh * sizeof *lab_copy);
  /* tout est pret pour lancer l'automate sur l'ensemble des cellules */
  for(i = 0; i < wh; ++i) {
    if(_is_explored(lab[i]) || lab[i] == WALL)
      continue;
    /* on est donc dans PATH ou EMPTY */
    /* récupérons l'abscisse et l'ordonnée de la cellule à l'indice i */
    int x = i % w, y = i / w; /* à retrouver dans le premier support de
        cours (intro) */
    /* l'abscisse et l'ordonnée du voisin (neighbor), un j pour les
       parcourir */
    int nx, ny, j;
    /* les 4 directions pour le voisinage en 4-connexité */
    const int d[][2] = { /* est */ { 1,  0 }, /* nord */  { 0,  1 },
       /* ouest */ {-1,  0 }, /* sud */   { 0, -1 } };
    for(j = 0; j < 4; ++j) {
      nx = x + d[j][0];
      ny = y + d[j][1];
      if( _in_screen(nx, ny, w, h) && _is_explored(lab_copy[ny * w + nx]) ) {
  lab[i] = _make_cost(_is_path(lab[i]) ? PATH : EXPLORED, _get_cost(lab_copy[ny
      * w + nx]) + 1);
  if(_is_path(lab[i]))
    gl4duwIdleFunc(next_cellular_automaton_step /* NULL si vous ne l'avez pas
        encore */);
      }
    }
  }
  free(lab_copy);
  gl4dpScreenHasChanged();
}
```

La Figure 6 illustre ce qui est attendu après chaque partie.

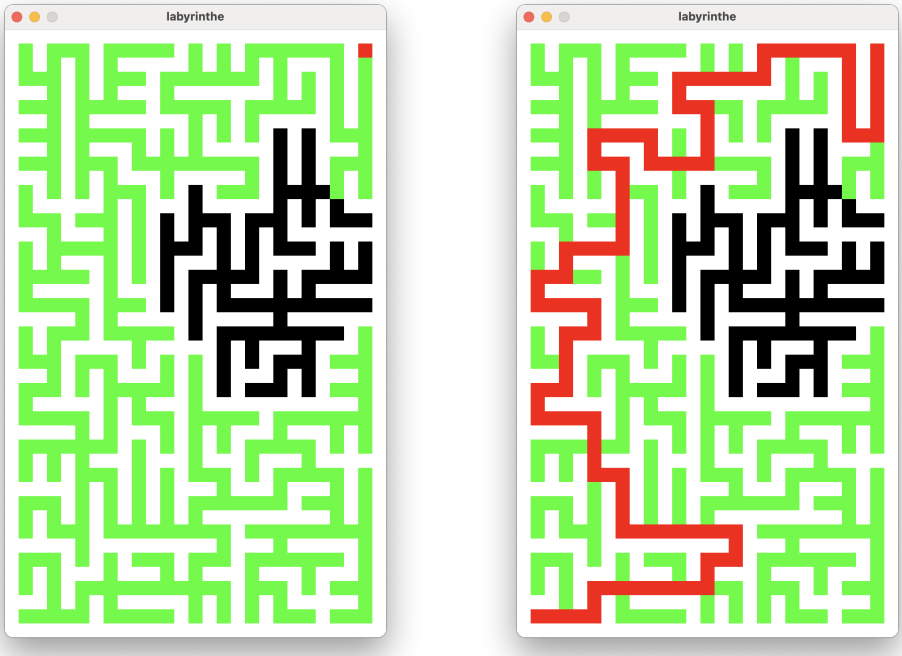


FIGURE 6 – Exemple de résultats attendus.