Labyrinthes
Construction et exploration

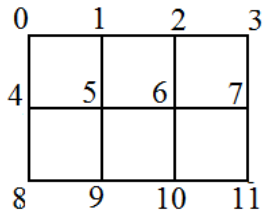
Un labyrinthe est un ensemble complexe de chemins tortueux, à embranchements multiples, dans lequel on peut tourner en rond et se perdre. Il existe un point d'entrée et aussi une issue qu'il convient d'atteindre, cette dernière pouvant être confondue avec le point d'entrée. On peut aussi placer en un certain endroit un objet qu'il s'agit d'atteindre. Dans tous les cas, on doit trouver un moyen d'explorer le labyrinthe en passant partout de façon systématique, du moins jusqu'à l'issue finale, en évitant de refaire plusieurs fois le même chemin ou de tourner en rond. Pour s'en sortir, on connaît le fil d'Ariane, les cailloux disposés sur son chemin par le Petit Poucet, ou la stratégie qui consiste à toujours longer les murs que l'on a à sa droite (ou à sa gauche si l'on préfère). Mais tout cela demande à être précisé. Nous allons procéder par étapes, en commençant par le cas le plus simple, et en finissant avec de véritables labyrinthes.

1. Un chemin tortueux, à défaut de labyrinthe

La méthode exposée ici va aussi être tortueuse, puisqu'elle va nous amener d'abord à construire des arbres couvrants pour un graphe, puis à construire un pseudo-labyrinthe où il suffit d'avancer sans jamais rebrousser chemin pour arriver à la sortie, et enfin à faire un pavage de dominos sur un carré.[1]

1.1. Arbre couvant et algorithme de Wilson

Considérons un graphe connexe non orienté avec ses sommets et ses arêtes. Par définition un arbre[2] couvrant de ce graphe a pour nœuds tous les sommets du graphe, et comme arêtes une partie de ses arêtes. Pour construire un arbre couvrant aléatoirement, utilisons l'algorithme de Wilson[3]. On commence par se donner sa racine, en prenant un sommet quelconque du graphe. Ce point est colorié en noir, par exemple. Puis on prend un sommet du graphe au hasard, et on lance un chemin au hasard sur le graphe, en évitant tout cycle, et en recommençant jusqu'à ce que le chemin atterrisse sur le point noir de la racine. On colorie alors ce chemin, avec les sommets concernés, en noir. Puis on prend un nouveau sommet non colorié en noir, et on lance à partir de lui un chemin sans cycle jusqu'à tomber sur un point noir. On tombe alors sur la racine ou sur le chemin déjà tracé. On colorie à son tour ce chemin en noir. Et l'on continue de la même façon tant qu'il existe un sommet non colorié du graphe à partir duquel on mène un chemin jusqu'à ce qu'il s'accroche à un sommet déjà noir. A la fin, tous les chemins obtenus, qui ne présentent aucun cycle, forment un arbre couvrant.



Appliquons cela à un graphe en forme de quadrillage, délimité par un rectangle avec N points en longueur et M points en largeur, comme par exemple sur le dessin ci-contre avec $N = 4$, et $M = 3$. Les $M \times N$ sommets du graphe sont numérotés de 0 à $MN - 1$, comme indiqué sur le dessin. Lors du passage de la zone calcul à l'écran, la longueur des côtés des carrés passe de 1 à *pas*, et l'origine du repère-écran (*xorig*, *yorig*) est le point numéro 0 (avec l'axe des ordonnées vers le bas). On en déduit les coordonnées-écran (x_i, y_i) des sommets du graphe :

[1] Nous utilisons ici le document de R. Kenyon et W. Werner, *Pavages, arbres et labyrinthes aléatoires*, *Images des maths*, 2004.

[2] Rappelons qu'u arbre est un graphe connexe sans cycles, qu'on lui donne ou non une racine.

[3] D.B. Wilson, *Generating random spanning trees more quickly than the cover time*, Proc. 28th ACM 296-303, 1996.

```
On se donne M et N, et leur produit est mis dans MN
for(i=0;i<MN;i++) { x[i]=xorig+pas*(i%N); y[i]=yorig+pas*(i/N);
                    filldisc(x[i],y[i],4,red);  /* dessin des sommets */
                  }
```

Il s'agit ensuite d'enregistrer le graphe en machine, en construisant la liste des voisins *v*[*i*][*j*] de chaque sommet *i*, ainsi que le nombre de ses voisins *nbv*[*i*]. Pour ce faire, on considère que tous les sommets autres que ceux de la première ligne (*i* – *N* >= 0) ont un voisin au-dessus d'eux, que ceux qui ne sont pas sur la bordure gauche (*i*%*N* != 0) ont un voisin à gauche, et de mêle avec les deux autres voisins.

```
for(i=0;i<MN;i++)
  { k=0;
    if (i-N>=0) v[i][k++]=i-N;
    if (i%N!=0) v[i][k++]=i-1;
    if (i+N<MN) v[i][k++]=i+N;
    if ((i+1)%N !=0) v[i][k++]=i+1;
    nbv[i]=k;
  }
for(i=0;i<MN;i++)  for(k=0;k<nbv[i];k++)  /* dessin des arêtes du graphe */
linewithwidth(x[i],y[i],x[v[i][k]],y[v[i][k]],0,green);
```

Passons maintenant à la construction de l'arbre couvrant aléatoire. On choisit par exemple comme racine *r* le point numéro 0, et l'on considère qu'on a fini de l'utiliser, en faisant *fini*[*r*] = 1, les valeurs de *fini*[] pour les autres points étant à 0. Le nombre *nbfinis* de sommets que l'on a fini d'utiliser passe à 1. Puis on lance une boucle *while* qui s'arrêtera lorsque tous les sommets seront utilisés (*nbfinis* = *MN*). A chaque fois on choisit au hasard un point *i0* non encore utilisé, à partir duquel on lance un chemin par le biais de la fonction *branche*(*i0*), et l'on dessine la branche correspondante de l'arbre couvrant, en utilisant les sommets de la branche qui sont enregistrés dans un tableau *a*[], et au nombre de *compteur* + 1 en comptant le dernier point s'accrochant à la partie de l'arbre déjà construite. Remarquons que le point *i0* constitue une feuille de l'arbre, provisoirement au moins.

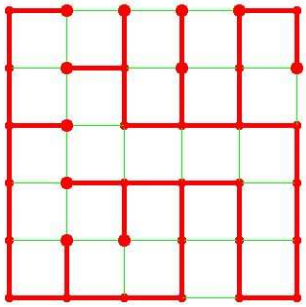
```
r=0; fini[r]=1;nbfinis=1;
while (nbfinis!=MN)
  { do {i0=rand()%MN;} while (fini[i0]==1);
     branche(i0); filldisc(x[a[0]],y[a[0]],6,red);
     for(k=0;k<compteur;k++) fini[a[k]]=1;
     for(k=0;k<compteur;k++)  linewithwidth(x[a[k]],y[a[k]],x[a[k+1]],y[a[k+1]],2,red);
  }
```

La fonction *branche*(*i*) part du sommet *i* et prend un de ses voisins au hasard. A partir de ce nouveau sommet, un voisin est choisi au hasard, et ainsi de suite jusqu'à ce que l'on arrive sur un sommet marqué comme fini. Mais on doit éviter qu'au cours de la construction du chemin on ne tombe sur un sommet déjà obtenu sur ce chemin, ce qui produirait un cycle. Lorsqu'un tel cas se présente, on décide d'effacer le cycle, en relançant le chemin à partir du point où commençait ce cycle. On utilise pour cela la fonction *oudejavu*(*v*) qui ramène – 1 si le sommet *v* n'a pas déjà été trouvé sur la branche en construction, et sinon qui ramène l'indice initial du sommet *v* au début du cycle, dans le tableau *a*[]. La variable *compteur* (déclarée en global) recommence à augmenter à partir de là.

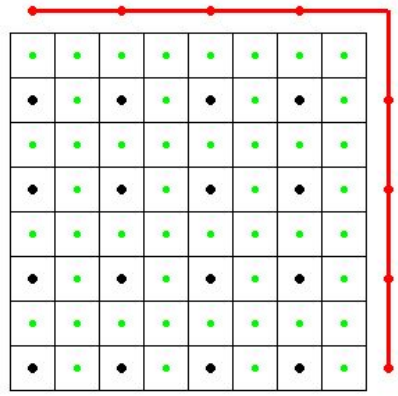
```
void branche(int i)
{ int h,j, voisin,sommet;
   compteur=0; a[0]=i;
  for(;;)
    { sommet=a[compteur];
      h=rand()%nbv[sommet]; voisin=v[sommet][h];
      if (fini[voisin]==1)
        {  compteur++; a[compteur]=voisin; nbfinis+=compteur ; break;}
```

```
      if (oudejavu(voisin)==-1) { compteur++;  a[compteur]=voisin;}
      else compteur=oudejavu(voisin);
    }
}

int oudejavu(int v)
{ int j;
  for(j=compteur;j>=0;j--)
  if (v==a[j]) return j;
  return -1;
}
```

Voici un exemple d'arbre couvrant pour $M = N = 6$



1.2. Une double arborescence



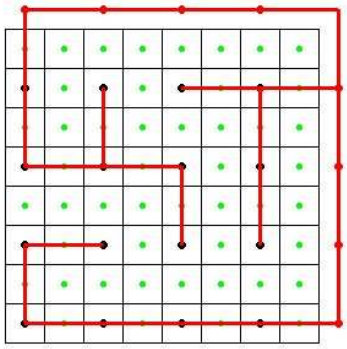
Prenons un quadrillage carré avec N points sur une longueur. Ce nombre N doit être pair. Les N^2 points de coordonnées (xe, ye) sont numérotés de gauche à droite et de bas en haut. Ils constituent les centres des petits carrés du quadrillage, de côté *pas* sur l'écran. Parmi les N^2 points, on privilégie ceux qui sont à coordonnées toutes les deux paires. Ils sont au nombre de $N^2/4$ et on les colorie en noir, comme sur le dessin ci-contre, avec $N = 8$. A leur tour, ces points (x_i, y_i) sont numérotés de gauche à droite et de bas en haut, avec i de 0 à $N^2/4 - 1$. On trace aussi à l'extérieur, le long des frontières nord et est, les points eux aussi à coordonnées paires situés à la distance $2\times pas$ des points noirs correspondants de la grille (rappelons que N doit être pair). Ces points extérieurs sont coloriés en rouge. La fonction *dessin*() se charge de dessiner cette grille de points et de petits carrés.

```
void dessin(void)
{  int i,xe,ye;
for(i=0;i<NN;i++) /*  grille carrée  N*N, le point numéro 0 étant en bas à gauche  */
   { xe=xorig+pas*(i%N); ye=yorig-pas*(i/N);
     filldisc(xe,ye,2,green);
     carre(xe-pas/2,ye-pas/2,xe+pas/2,ye+pas/2,black);
   }
 for(i=0;i<NN/4;i++) /* Points à coordonnées paires en noir */
   { x[i]=xorig+pas*((2*i)%N); y[i]=yorig-pas*2*((2*i)/(N));
     filldisc(x[i],y[i],3,black);
   }
 for(i=0;i<N/2;i++)  filldisc(xorig+pas*N,yorig-pas*2*i,3,red); /* Points racines au nord et à l'est */
 for(i=0;i<N/2;i++)  filldisc(xorig+pas*2*i,yorig-pas*N,3,red);
 linewithwidth(xorig+pas*N,yorig,xorig+pas*N,yorig-pas*N ,1,red);
 linewithwidth(xorig,yorig-pas*N,xorig+pas*N,yorig-pas*N ,1,red);
}
```

Considérons maintenant la grille des points noirs ainsi que les points rouges associés, et construisons les arbres couvrants dont les racines sont ces points rouges, ce qui donne une forêt d'arbres, ou si l'on préfère un arbre unique si l'on relie les points rouges entre eux et que l'on considère qu'ils forment alors une racine unique. Les branches des arbres ainsi obtenus, coloriées en rouge, vont constituer une partie des murs d'un labyrinthe. La fonction *premiersmurs*() se charge de cette construction. En la circonstance on aménage la fonction *branche*() du paragraphe précédent. Cette fonction est lancée à partir d'un point qui n'est pas rouge, appelé *feuille* dans le programme. A partir d'un sommet du chemin en cours de construction, on détermine les coordonnées de ses quatre voisins (à la distance $2\times pas$) en éliminant éventuellement un point extérieur colorié en blanc, qui ne peut pas être un voisin dans le quadrillage, et l'on prend un de ces voisins au hasard. Si ce voisin est rouge, on arrête la construction de la branche, car elle vient de s'accrocher à une autre branche ou à une racine rouge. Sinon on prend ce voisin sous réserve qu'il ne forme pas un cycle sur le chemin, et on relance le cheminement par voisinage. Remarquons que le nombre de points sur une branche, sans compter le dernier point d'accroche, est *compteur* + 1, leurs numéros étant enregistrés dans le tableau *a*[*k*] avec *k* allant de 0 à *compteur*.

```
void premiersmurs(void)
{ int i,k,j,r;
  nbfinis=0;
  while(nbfinis<NN/4)
   { do r=rand()%(NN/4);   while (getpixel(x[r],y[r])==red);
     branche(r);
     for(k=0;k<compteur;k++) linewithwidth(x[a[k]],y[a[k]],x[a[k+1]],y[a[k+1]],1,red);
     nbfinis+=compteur+1;
   }
 }

void branche(int feuille)
{ int h,j,sommet,voisin[4];
   compteur=0; a[0]=feuille;
   for(;;)
   { sommet=a[compteur]; /* sommet qui court en partant de feuille */
     xvoisin[0]=x[sommet]+2*pas; yvoisin[0]=y[sommet]; /* coordonnées des quatre voisins */
     xvoisin[1]=x[sommet]; yvoisin[1]=y[sommet]-2*pas;
     xvoisin[2]=x[sommet]-2*pas; yvoisin[2]=y[sommet];
     xvoisin[3]=x[sommet]; yvoisin[3]=y[sommet]+2*pas;
     do h=rand()%4;  while (getpixel(xvoisin[h],yvoisin[h])==white);
     if (getpixel(xvoisin[h],yvoisin[h])==red) /* dernier trait de la branche */
        { linewithwidth(x[sommet],y[sommet], xvoisin[h],yvoisin[h],1,red);
          break;
        }
     else
        { voisin[0]=sommet+1; voisin[1]=sommet +N/2; /* numéros des voisins */
          voisin[2]=sommet-1; voisin[3]=sommet -N/2;
          if (oudejavu(voisin[h])==-1)     { compteur++;  a[compteur]=voisin[h]; }
          else compteur=oudejavu(voisin[h]);
        }
   }
}
int oudejavu(int v)
{ int j;
  for(j=compteur;j>=0;j--)
  if (v==a[j]) return j;
  return -1;
}
```

On trouve ainsi les premiers murs rouges du labyrinthe futur.



Cela étant fait, on va maintenant s'occuper des points restants de la grille, jusqu'à présent coloriés en vert. Commençons par tracer deux bordures extérieures ouest et sud en bleu (*figure 1 à gauche*). Puis joignons chaque point vert à ses voisins sous réserve qu'ils soient verts ou bleus, c'est-à-dire non coloriés en rouge. Les chemins ainsi formés sont coloriés en bleu, et il s'agit d'une forêt d'arbres, car s'il existait un cycle, celui-ci encerclerait des points rouges, ce qui est impossible à cause de la connexité de l'arborescence rouge. Il en découle la fonction *deuxiemesmurs*().[4]

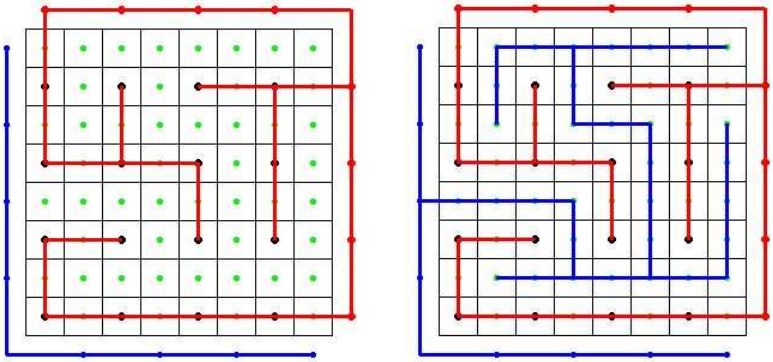


Figure 1 : *A gauche*, les bordures extérieures en bleu, et *à droite* l'arborescence bleue en complément de la rouge

```
void deuxiemesmurs(void)
{ int i,j;
for(i=0;i<N/2;i++)  filldisc(xorig-pas,yorig-pas-pas*2*i,2,blue); /* racines extérieures bleues*/
for(i=0;i<N/2;i++)  filldisc(xorig+pas+pas*2*i,yorig+pas,2,blue);
 linewithwidth(xorig-pas,yorig+pas,xorig-pas,yorig-pas*(N-1) ,1,blue);
 linewithwidth(xorig-pas,yorig+pas,xorig+pas*(N-1),yorig+pas ,1,blue);
 for(i=0;i<NN;i++)   { x[i]=xorig+pas*(i%N); y[i]=yorig-pas*(i/N);}
 for(i=0;i<NN;i++) if (getpixel(x[i],y[i])!=red) /* on prend chaque point non rouge du quadrillage */
   { for(j=0;j<4;j++)
       { xvoisin[0]=x[i]+pas; yvoisin[0]=y[i];
         xvoisin[1]=x[i]; yvoisin[1]=y[i]-pas;
         xvoisin[2]=x[i]-pas; yvoisin[2]=y[i];
         xvoisin[3]=x[i]; yvoisin[3]=y[i]+pas;
         if (getpixel(xvoisin[j],yvoisin[j])!=red) /* on prend les voisins non rouges */
            linewithwidth(x[i],y[i],xvoisin[j],yvoisin[j],1,blue); /* jonction en bleu */
       }
   }
}
```

[4] On aurait pu aussi bien commencer par prendre les points à coordonnées toutes deux impaires, ce qui aurait donné l'arborescence bleue, puis on aurait pris son complément en rouge.

1.3. Parcours du labyrinthe

Les traits rouges et bleus sont maintenant considérés comme des murs d'un labyrinthe, avec la présence de deux ouvertures, l'une en haut à gauche, l'autre en bas à droite. Les couloirs intermédiaires s'inscrivent entre les murs rouges et les murs bleus. Si l'on part de l'ouverture du haut, le fait de longer les murs bleus à gauche (ou les murs rouges à droite) constitue un parcours d'arbre dont chaque arête est parcourue dans un sens puis dans l'autre, sauf les arêtes frontières extérieures qui ne sont parcourues que dans un sens. On arrive ainsi à la sortie sans jamais risquer d'avoir à faire des choix, et encore moins de se perdre dans un cycle. Il suffit d'avancer du départ à l'arrivée, en suivant un chemin tortueux puisqu'il remplit le carré global. En ce sens il ne s'agit pas d'un véritable labyrinthe.

Pour construire ce chemin, on fait appel à la fonction *exploration*(), qui appelle à son tour la fonction récursive *chemin*() à partir du point de départ. On constate que le chemin suit des traits noirs du quadrillage des petits carrés initiaux. Lorsque l'on est en un point (*xe*, *ye*) du chemin, il suffit de prendre les quatre voisins situés à une distance de *pas*/2 (on devra choisir *pas* pair). Comme on ne rebrousse jamais chemin, l'un des quatre voisins ne peut pas être pris, celui qui est dans la direction *interdit*. Sur les trois voisins restants, deux tombent sur les murs rouge et bleu, et le troisième tombe sur un trait noir du quadrillage. C'est dans cette dernière direction que l'on va aller, et l'on avance alors sur la longueur *pas* (et non pas *pas*/2). Par la même occasion on connaît quelle est la direction interdite pour le nouveau point obtenu, et la fonction *chemin*() se rappelle sur ce point avec la variable *interdit* correspondante. Au départ, la fonction chemin est appelé sur le point de départ, avec comme direction interdite la direction 1, c'est-à-dire la direction nord (0 est la direction est, 2 la direction ouest, et 3 la direction sud). Chaque nouveau point obtenu lors du parcours est colorié en noir, tandis que son prédécesseur est colorié en gris. De la sorte, le cheminement laisse apparaître une traînée grise derrière lui (*figure 2*).

```
void exploration(void)
{
   filldisc(xorig-pas/2,yorig-pas*(N-1)-pas/2,pas/4,black);  /* on colorie le point de départ en noir */
   chemin(xorig-pas/2,yorig-pas*(N-1)-pas/2,1); /* chemin à partir du point de depart */
}

void chemin(int xe, int ye, int interdit)
{ int i;
   filldisc(xe,ye,pas/4,black);SDL_Flip(screen); SDL_Delay(10);
   if (!(xe==xorig+pas*(N-1)+pas/2 && ye==yorig+pas/2)) /* test d'arrêt */
   for(i=0;i<4;i++) if (i!=interdit)
        { xvoisin[0]=xe+pas/2; yvoisin[0]=ye;
          xvoisin[1]=xe; yvoisin[1]=ye-pas/2;
          xvoisin[2]=xe-pas/2; yvoisin[2]=ye;
          xvoisin[3]=xe; yvoisin[3]=ye+pas/2;
          if (getpixel(xvoisin[i],yvoisin[i])==black)
             {  filldisc(xe,ye,pas/4,gray);
                chemin(xvoisin[i],yvoisin[i],(i+2)%4);
             }
        }
}
```

Le programme principal consiste à appeler les fonctions précédentes, soit :

```
dessin();
premiersmurs();
deuxiemesmurs();
exploration();
```

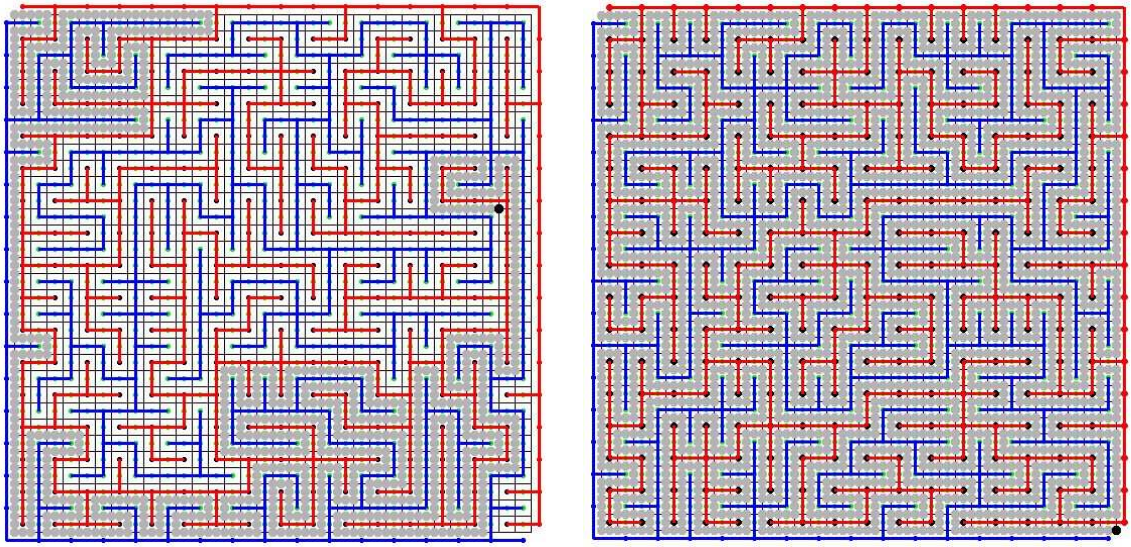
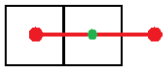


Figure 2 : *A gauche* le cheminement en cours, *à droite* le chemin du début à la fin

1.4. Prolongement : Pavage aléatoire d'un carré par des dominos

Reprenons l'arborescence rouge telle qu'elle a été précédemment construite. A partir de ce qui va être une feuille de l'arbre, on trace un chemin allant d'un point à coordonnées paires à un voisin à coordonnées paires aussi, à une distance de 2 pas, suivant une certaine direction.



Comme indiqué sur le dessin ci-contre, où la direction est 0, on peut placer un domino formé de deux petits carrés unités accolés (de côté *pas*) dont le point rouge est le centre du premier carré. Chaque branche de l'arborescence rouge peut alors être remplacée par une succession de dominos (*figure 3*).

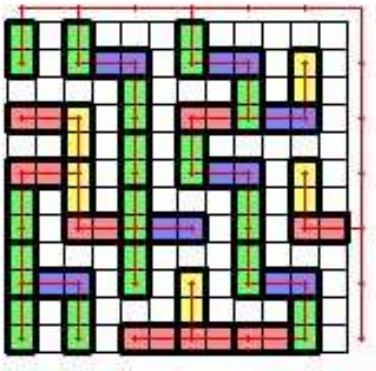


Figure 3 : Remplissage de l'arborescence rouge par des dominos. Ceux-ci sont de quatre couleurs, selon leur direction

Il reste à faire de même avec l'arborescence bleue, que l'on va aussi couvrir de dominos en succession, ce qui remplira le carré avec des dominos ainsi placés de façon aléatoire. Mais celle-ci n'a pas été construite en partant des feuilles, comme on l'a fait avec l'arborescence rouge. Il convient de déterminer où sont situés les points formant les feuilles de l'arborescence bleue. Il s'agit de ceux qui ont trois voisins, à un pas de distance, non coloriés en blanc. On peut alors placer des dominos en ces points, ce qui donne un remplissage partiel. On trouve encore des points entourés de trois voisins non blancs. On place ici encore des dominos. Et l'on répète cela autant de fois qu'il faut pour remplir totalement la zone (*figure 4*).

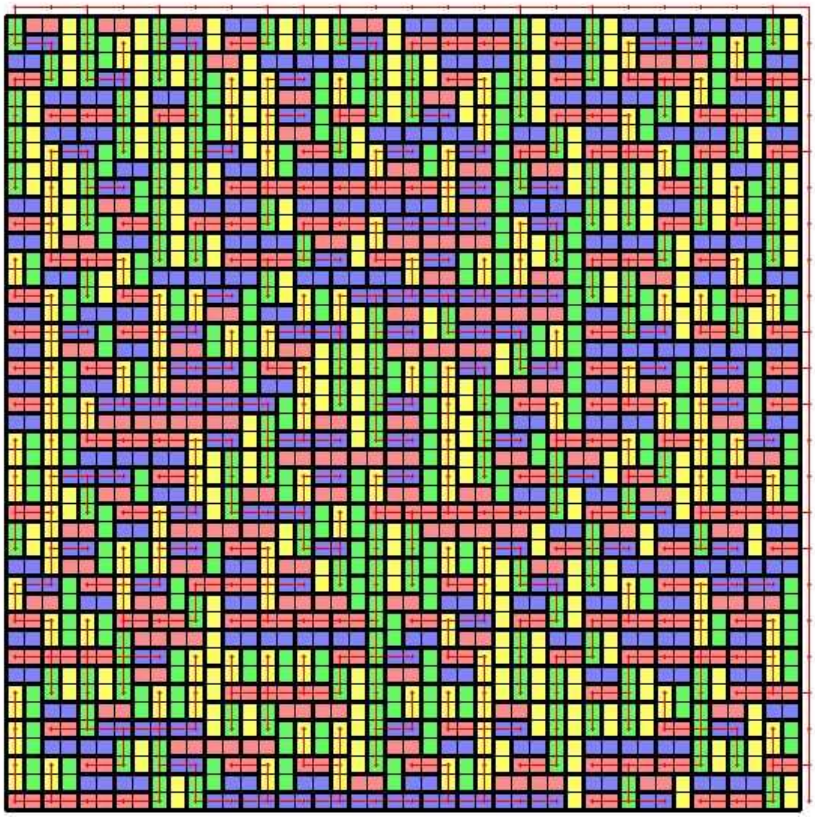


Figure 4 : Pavage aléatoire d'un carré de côté pair par des dominos

1.5. Un exemple antique

Voici une mosaïque grecque antique qui est censée représenter le labyrinthe conçu par Dédale, avec le Minotaure en son centre. C'est le héros grec Thésée qui avait pour mission de tuer le Minotaure. Tel que le dessin est fait, avec l'entrée en bas au centre, Thésée n'avait qu'à suivre le chemin tortueux qui finissait par le mener jusqu'au Minotaure, et il pouvait ensuite revenir dans l'autre sens sans risque de se perdre. Dans ces circonstances, le fil d'Ariane qu'il était censé dérouler à partir de l'entrée pour pouvoir revenir sans peine ne servait à rien.

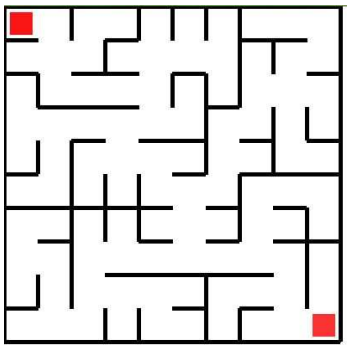
2. Un labyrinthe simplifié

2.1. Construction[5]

Prenons un quadrillage formé de carrés de côté unité, le tout délimité par un grand carré de longueur *L*. Les murs du labyrinthe vont être des traits tracés sur le quadrillage, et la bordure du grand carré constitue le mur d'enceinte. Seul ce mur extérieur est construit au départ, puis on se place à l'intérieur, et l'on prend au hasard des vecteurs successifs, de longueur unité, orientés dans l'une des quatre directions est, nord, ouest, sud, tels que leur origine soit un point du quadrillage, c'est-à-dire un sommet d'un petit carré, et que leur extrémité soit sur un mur. Avec les murs dessinés en noir et les points à l'intérieur tous en blanc au début, il s'agit de chercher à joindre un point blanc et un point noir. Chaque fois qu'un tel vecteur est trouvé, on le dessine en noir, et notamment son origine passe de

[5] Je reprends ici la méthode exposée dans mon livre *Combien ? Mathématiques appliquées à l'informatique, tome 3 : graphes* (ou *Mathematics for computers and informatics*, en anglais).

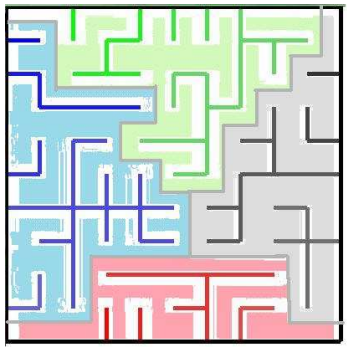
blanc à noir. Au début, le seul mur existant est le mur d'enceinte, puis de petits bouts de mur vont s'accrocher au mur d'enceinte, puis d'autres bouts vont s'accrocher aux murs existants. Comme ce procédé exclut la formation de cycles, puisqu'aucun point noir n'est jamais joint à un point noir, on obtient des arbres dont la racine est accrochée à la bordure extérieure. La construction s'arrête quand tous les points intérieurs du quadrillage sont colorés en noir. On est alors sûr que tous les arbres-murs ont une distance unité qui les sépare, ce qui forme des couloirs intercalaires constituant le labyrinthe.



Il reste à choisir un point de départ *A* pour circuler dans le labyrinthe, ainsi qu'un point d'arrivée *B*. Nous avons choisi les deux points situés aux antipodes du grand carré, le point de départ étant situé en haut à gauche. Ces points sont dessinés sous forme de deux petits carrés rouges (*figure 5*).

Figure 5 : Le labyrinthe à l'intérieur du mur d'enceinte, et les points d'entrée et de sortie en rouge

2.2. Exploration



On distingue quatre types d'arbres-murs, selon que leur racine est à l'ouest, au sud, à l'est ou au nord, comme indiqué sur la *figure ci-contre* où ils sont coloriés différemment. Le grand carré est ainsi découpé en quatre zones, dont les frontières séparent essentiellement les arbres verts et bleus, les arbres bleus et rouges, les arbres rouges et noirs, et ceux qui sont noirs et verts (*voir figure*). Ces frontières constituent des parcours de cimes des arbres. Par la même occasion, on vient de trouver un chemin menant du point d'entrée *A* au point d'arrivée *B*, en suivant les cimes des arbres bleus puis des arbres rouges. A leur tour, les couloirs entre les murs entourent chacun des arbres-murs, ils forment des chemins capillaires allant de la cime des arbres vers leur racine. A partir du chemin de cimes entre *A* et *B* partent des chemins se terminant tous sur le mur d'enceinte. Finalement l'ensemble des chemins du labyrinthe a une structure d'arbre dont la racine est le point *A* et dont les feuilles sont sur les murs d'enceinte. Il en découle qu'il existe un chemin unique menant de *A* à *B*, celui qui longe les cimes des arbres bleus puis rouges (ou encore celles des arbres verts puis noirs).[6] C'est pour cette raison que nous parlons de labyrinthe simplifié, car il s'agit d'une simple exploration d'arbre, d'une partie de l'exploration plus précisément, puisque l'on s'arrête dès que l'on atteint le point *B*.[7]

On sait comment parcourir un arbre, en choisissant par exemple d'aller à droite lorsque plusieurs bifurcations se présentent, et en faisant demi-tour lorsqu'on atteint une feuille de l'arbre. Il reste à faire de même dans notre labyrinthe. Pour cela considérons un mobile en forme de carré, caractérisé par sa position (*x*, *y*) et dirigé dans une certaine direction, celle qu'il a devant lui. A partir de cette direction, le mobile peut soit garder cette direction, soit faire un quart de tour à droite, soit faire un quart de tour à gauche, soit faire demi-tour. Pour respecter les conditions de l'exploration, il va privilégier de tourner à droite, mais s'il tombe sur un mur, il choisira d'aller devant, et s'il tombe encore sur un mur, il ira à gauche. Enfin, s'il tombe sur un cul-de-sac, il fera demi-tour.

[6] Quand nous parlons de chemin, nous voulons dire route, ou chemin élémentaire, c'est-à-dire un chemin qui ne repasse pas sur lui-même.

[7] Si l'on veut parcourir l'autre partie de l'arbre, on peut aller de *B* vers *A* en privilégiant d'aller à droite.

De la sorte, notre mobile va parcourir toute la partie droite de l'arbre d'exploration jusqu'à son arrivée en *B*. Sur la *figure 6*, on voit son cheminement sur cette partie d'arbre, sous forme de petits carrés rouges et jaunes, les jaunes indiquant qu'il rebrousse chemin. En éliminant les allers-retours sur les chemins se terminant sur des feuilles de l'arbre d'exploration, il reste le chemin unique menant de *A* à *B* (*figure 6 à droite*).

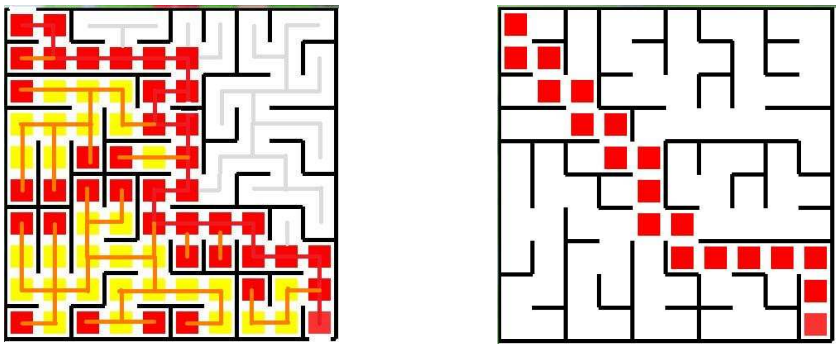


Figure 6 : *A gauche*, parcours des couloirs du labyrinthe, jusqu'à l'arrivée en *B*. Il s'agit de l'exploration d'une partie de l'arbre des chemins, en privilégiant d'aller à droite. *A droite*, le chemin unique menant de *A* à *B*.

2.3. Programmation

Commençons par dessiner un carré de côté de longueur *L*, en fait de longueur *pas*L* sur l'écran, la longueur unité du quadrillage ayant une longueur *pas* en pixels sur l'écran (par exemple *pas* = 10). Pour simplifier, un sommet de ce carré est le point (0, 0) sur l'écran, soit le point en haut à gauche de la fenêtre écran.

```
rectangle(0,0,pas*L,pas*L,2,black);
```
[8]

A ce stade, il y a $L + 1$ points du quadrillage sur chaque côté de ce carré, et par suite $L - 1$ points sur chaque longueur à l'intérieur strict du carré. Au départ les $(L - 1)^2$ points à l'intérieur strict du carré sont tous en blanc, et l'on met la variable *nbfini* à 0 (il suffit de déclarer cette variable en global), ce qui indique que tous les points intérieurs sont en blanc. Puis on lance une boucle *for*(;;).Chaque fois qu'un de ces points blancs sera colorié en noir, avec la construction d'un mur passant sur ce point, la variable *nbfini* augmentera de 1, et lorsque *nbfini* atteindra $(L - 1)^2$, cela voudra dire que la construction du labyrinthe est finie. Cela constitue l'arrêt de la boucle *for*(;;).

A chaque fois que la boucle est lancée, on prend un point (*x*, *y*) au hasard, jusqu'à obtenir un point blanc. Puis on prend une direction au hasard parmi les quatre possibles, et l'on prend le vecteur correspondant d'origine (*x*, *y*). Si l'extrémité de ce vecteur n'est pas noire, rien ne se passe et l'on relance la boucle *for*(;;). Lorsque l'on finit par tomber sur une extrémité noire, on trace le trait noir joignant l'origine (*x*, *y*) à cette extrémité, ce qui donne un mur du labyrinthe. D'où la suite du programme de construction :

```
for(;;)
{    if (nbfinis==(L-1)*(L-1)) break;
     do   { x=rand()%(L+1);y=rand()%(L+1); }
     while(getpixel(pas*x,pas*y)==black);
     dir=rand()%4;
     dx=((dir+1)%2)*(1-dir);dy=(dir%2)*(dir-2);
      if (getpixel(pas*x+pas*dx,pas*y+pas*dy)==black)
```

[8] Il suffit de reprendre la fonction *rectangle*(), mais on a intérêt à remplacer les *line* par des *linewithwidth* pour donner aux murs une certaine épaisseur. Rappelons que les fonctions graphiques utilisées sont données sur mon site dans *graphical functions*.

```
        { linewithwidth(pas*x,pas*y,pas*x+pas*dx,pas*y+pas*dy,2,black);
          nbfinis++;
        }
  }
```

Passons maintenant à l'exploration. On commence par prendre le point de départ *A* de coordonnées écran (*xd*, *yd*), ainsi que le point d'arrivée (*pas*L- pas/*2, *pas*L-pas/*2). On décide aussi de donner au mobile la direction 3 (verticale vers le bas) au départ, celle qui est la plus à droite possible. Puis on dessine les carrés qui entourent ces deux points de départ et d'arrivée avec un certain rouge, notamment *rougeend* pour le point d'arrivée. La fonction *carre*() prend comme variables les coordonnées écran du point, ainsi qu'un nombre *q* qui indique que le côté du carré mesure 2*pas/q, et la couleur *c* de remplissage. Pour empêcher ce carré d'empiéter sur un mur, il est nécessaire de prendre *q* > 2, par exemple *q* = 3.

```
void carre(int ii, int jj , int q,int c)
{ int i,j;
  for(i=ii-pas/q;i<=ii+pas/q;i++)  for(j=jj-pas/q;j<=jj+pas/q;j++)
  putpixel(i,j,c);
}
```

On fait alors dans le programme principal :

```
xd=pas/2;yd=pas/2;
x=xd;y=yd;direction=3;
carre(x,y,3,reddebut);
carre(pas*L-pas/2,pas*L-pas/2,3,redend);
SDL_Flip(ecran); pause();
```

Puis on lance la boucle *for*(;;). Au début de chaque étape de la boule, le mobile orienté a pour coordonnées écran (*x, y*) et pour direction *direction*, et il s'agit de déterminer le point (*newx*, *newy*) qui va lui succéder, avec la direction *newdirection*. Cela étant fait, on actualisera ces trois variables en faisant *x* = *newx*, *y* = *newy* et *direction* = *newwdirection*, et la boucle repart pour trouver le point suivant du chemin. Comment passer d'un point du chemin au suivant ? Pour cela on a besoin de connaître les trois voisins concernés. Si à partir de *direction*, on fait un quart de tour à droite, la nouvelle direction est *right* = (*direction* + 3)%4, et le point voisin se déduit du point actuel par une variation de coordonnées calcul (*dxdr*, *dydr*) que l'on peut calculer. Si l'on fait avancer le mobile devant lui, la nouvelle direction est *front* = *direction*, et les variations de coordonnées calcul sont (*dxde*, *dyde*). Si le mobile doit aller à gauche, soit *left* = (*direction* + 1)%4, les variations de ses coordonnées calcul sont (*dxga*, *dyga*). Pour respecter l'exploration qui privilégie d'aller à droite, on commence par choisir le quart de tour à droite. Si à la distance *pas*/2 du point où l'on est, on tombe sur un point blanc, on choisir d'aller dans cette direction, d'où le nouveau point (*newx*, *newy*) avec la direction *newdirection* = *right*. Mais si l'on tombe sur un point noir, c'est-à-dire sur un mur, on choisit d'aller en avant (*front*) si c'est possible. Sinon, on choisit de faire un quart de tour à gauche, en prenant la direction *left*, si c'est possible. Sinon, on est encerclé par trois murs, et l'on doit faire demi-tour, sans bouger, soit *newx* = *x*, *newy* = *y*, *newdirection* = (*direction* + 2)%4.

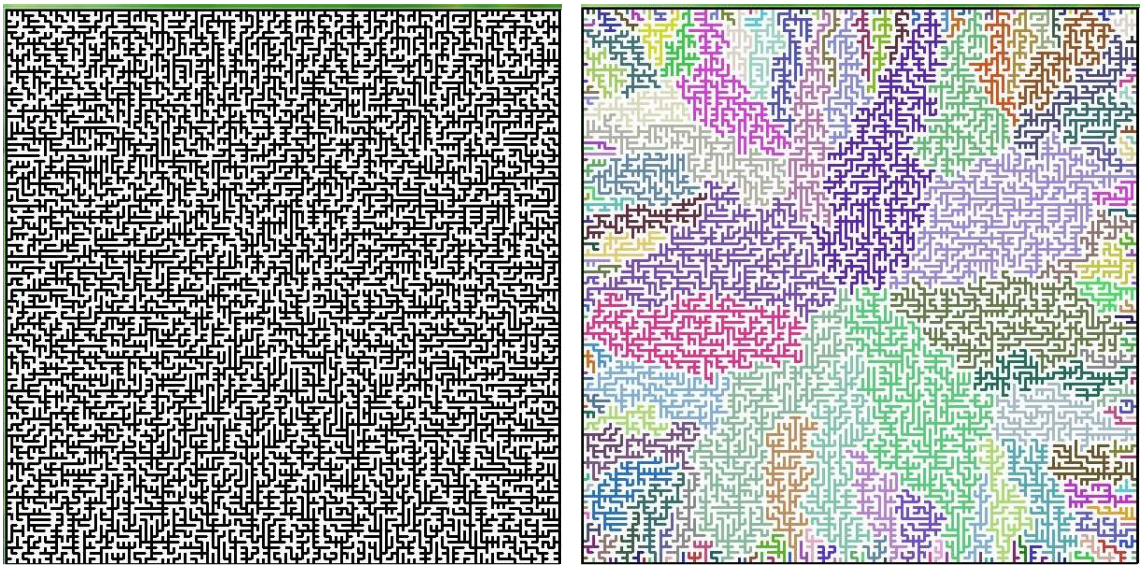
Une fois que ce nouveau point est trouvé, on teste alors si l'on tombe sur le point d'arrivée, avec sa couleur spécifique *redend*. Si tel est le cas, on arrête la boucle *for*(;;). Sinon, on décide de colorier le point où l'on est en rouge (en fait on colorie le carré autour du point) et le nouveau point en jaune, de façon provisoire. Mais pas exactement, car deux cas se présentent. Si le successeur était jusque là en rouge, cela veut dire que l'on est en train de rebrousser chemin, auquel cas on dessine le successeur en jaune, mais l'on efface le point où l'on est. Sinon, on est en marche avant, le point actuel est mis en rouge, et le successeur en jaune.

```
for(;;)
{
right=(direction-1+4)%4;   /* pré-calcul des trois voisins du point où l'on est */
```

```
dxdr=((right+1)%2)*(1-right);dydr=(right%2)*(right-2);
front=direction;
dxde=((front+1)%2)*(1-front);dyde=(front%2)*(front-2);
left=(direction+1)%4;
dxga=((left+1)%2)*(1-left);dyga=(left%2)*(left-2);

if (getpixel(x+pas*dxdr/2,y+pas*dydr/2)==blanc) /* choix prioritaire d'aller à droite */
    { newx=x+pas*dxdr;newy=y+pas*dydr;newdirection=right; }
else if (getpixel(x+pas*dxde/2,y+pas*dyde/2)==blanc) /* sinon devant */
    { newx=x+pas*dxde;newy=y+pas*dyde;newdirection=front;}
else if (getpixel(x+pas/2*dxga,y+pas/2*dyga)==blanc) /* sinon à gauche */
    { newx=x+pas*dxga;newy=y+pas*dyga;newdirection=left;}
else {newx=x;newy=y; newdirection=(direction+2)%4; } /* sinon demi-tour */

if (getpixel(newx,newy)==redend) { carre(x,y,3,red);break;}

if (getpixel(newx,newy)==red)
    { carre(newx,newy,3,yellow);
      carre(x,y,3,blanc);
     }

else {  carre(newx,newy,3,yellow);
         carre(x,y,3,red);
      }
SDL_Flip(ecran); x=newx;y=newy; direction=newdirection;
}
```

2.4. Un exemple de grand labyrinthe

On a pris un carré avec $L = 99$, les résultats sont donnés sur la *figure 7*.



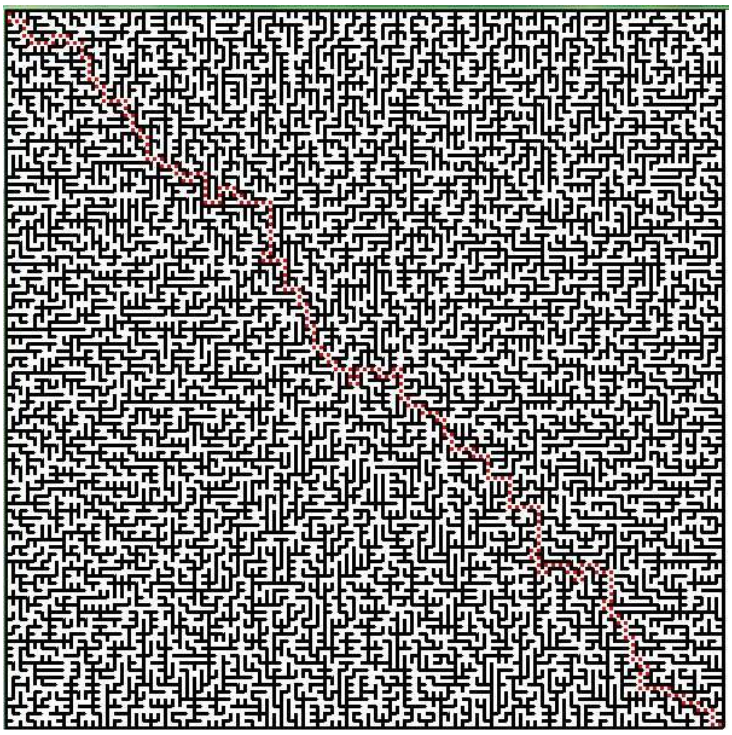


Figure 7 : *En haut à gauche* le labyrinthe, *à droite* les arbres qui se cachent sous le labyrinthe. *En bas* le chemin unique de *A* à *B*.

2.5. Variante

Pour mieux comprendre comment fonctionne l'algorithme précédent, il suffit de diminuer la taille des arbres-murs. Au lieu d'attendre qu'ils remplissent l'intérieur du carré, on arrête leur croissance avant, par exemple en stoppant la construction lorsque *nbfinis* = (*L* – 1)2 /2, ce qui laisse la zone centrale du carré vide. Mais le programme précédent continue de fonctionner. Le chemin de *A* à *B* se construit toujours en allant à droite de préférence, c'est-à-dire en longeant la cime des arbres dont les racines sont à l'ouest ou au sud (*figure 8*). Remarquons qu'en privilégiant le fait d'aller à gauche, on aurait un autre chemin allant de *A* à *B*, en longeant les arbres-murs dont les racines sont au nord et à l'est.

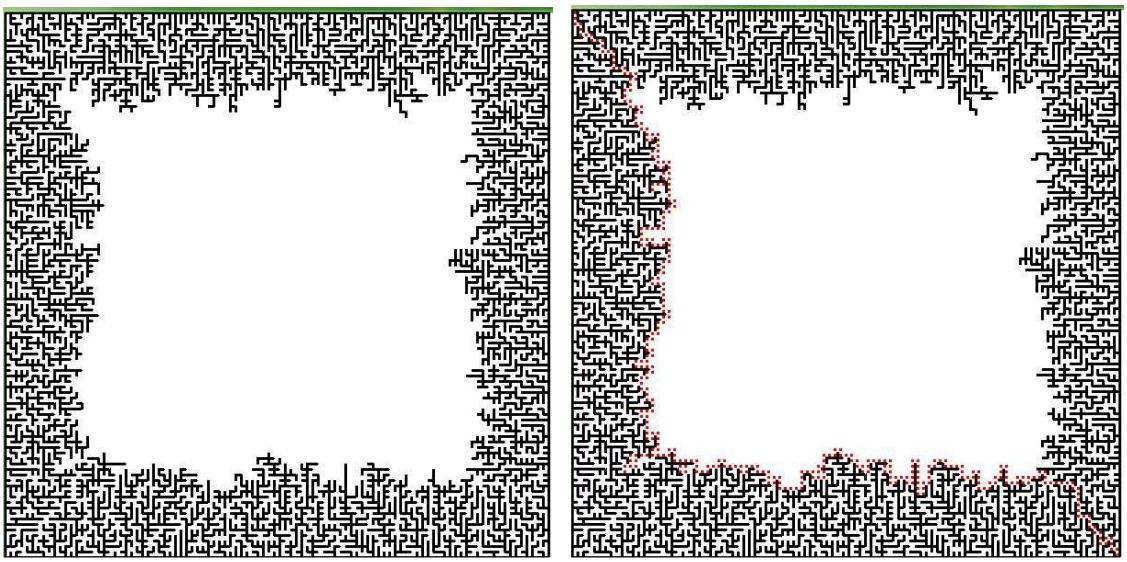


Figure 8 : *A gauche* le labyrinthe partiel, et *à droite* le chemin rouge qui longe les cimes

2.6. Amélioration : Un vrai labyrinthe

Au lieu de partir seulement du mur d'enceinte colorié en noir, rajoutons quelques points noirs pris au hasard à l'intérieur du carré. Ces points vont à leur tour servir de points d'ancrage autour desquels des arbres-murs vont se développer, et non plus seulement ceux qui s'accrochent au mur d'enceinte. On obtient alors un véritable labyrinthe, puisque des cycles peuvent se produire lors du parcours de *A* à *B*. Et il existe plusieurs chemins pouvant aller de *A* à *B*. Mais l'algorithme précédent continue de fonctionner : il privilégie le fait d'aller à droite et s'appuie toujours sur les cimes des arbres dont les racines sont sur les murs à l'ouest et au sud. On trouve ainsi un des chemins menant de *A* à *B* (*figure 9*).

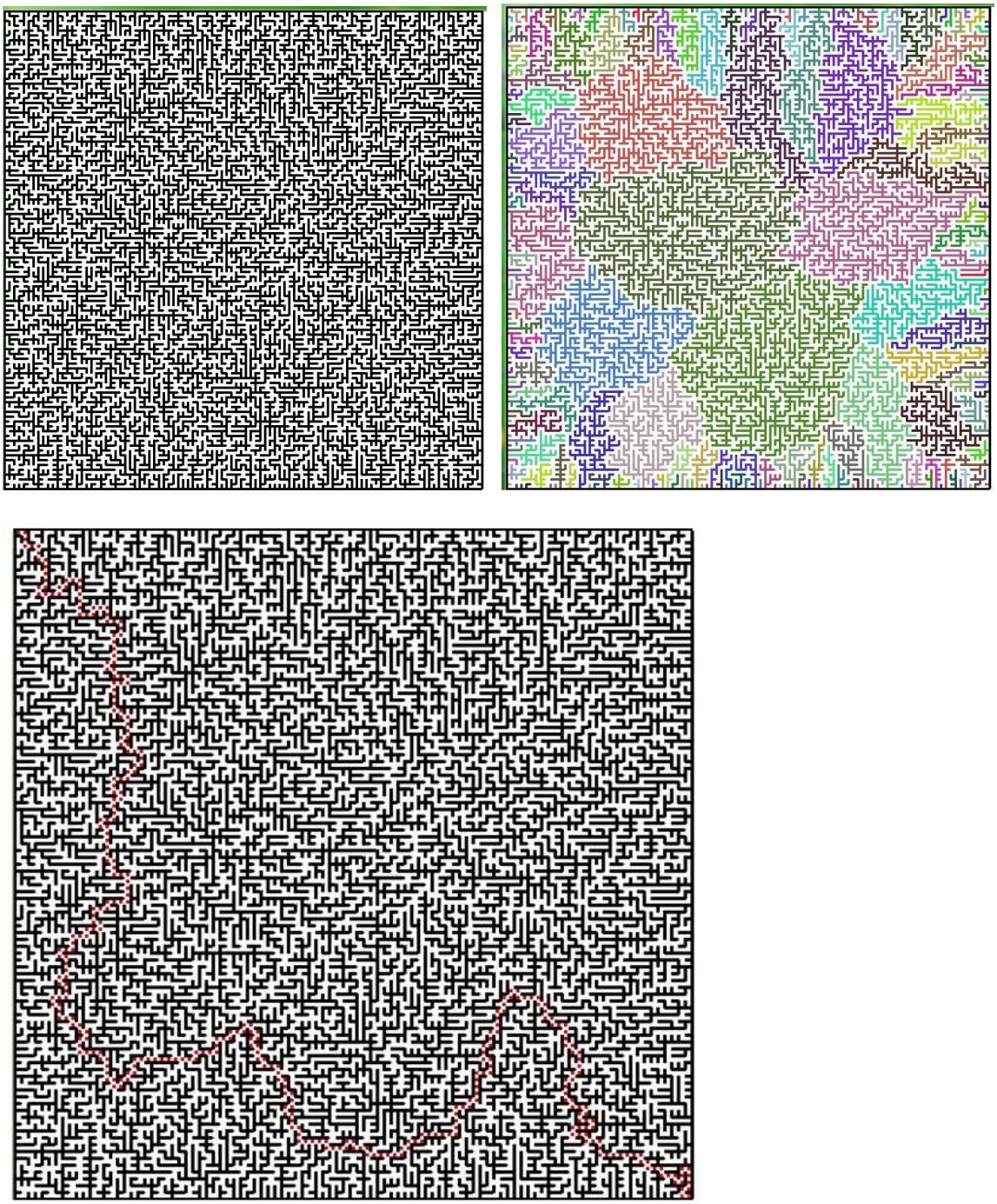


Figure 9 : *En haut à gauche*, le labyrinthe, *à droite* les arborescences, les unes ayant leurs racines sur les murs d'enceinte, les autres autour des points d'ancrage intérieurs. *En bas* le chemin de *A* à *B* obtenu par l'algorithme du choix d'aller à droite.

3. Le labyrinthe quelconque

Nous allons explorer le labyrinthe. Qu'entend-on par là ? Il va s'agir, à partir d'un point de départ, de parcourir tous les couloirs et surtout d'atteindre tous les carrefours, sans se perdre dans des boucles infinies, pour finalement se retrouver au point de départ. De cette façon, même si une sortie se trouve ailleurs qu'au point de départ, ou encore s'il s'agit d'arriver à un objet qui se trouve quelque part dans le labyrinthe, que ce soit à un carrefour ou au milieu d'un couloir, on sera sûr de les atteindre. Dans ces conditions, cette méthode d'exploration n'est pas forcément la plus rapide, mais elle a l'avantage d'être sûre, puisque l'on va passer partout sans se perdre.

3.1. De l'exploration d'un graphe en profondeur à l'exploration d'un labyrinthe

Tout commence par un problème classique : l'exploration d'un graphe en profondeur.[9] Telle que la programmation est faite sur machine, le parcours s'apparente un peu à celui d'un piéton, dans le sens où le piéton avance dans le graphe aussi profondément que possible,[10] en prenant la première route qui s'ouvre devant lui, tout en marquant les sommets sur lesquels il passe, afin d'éviter d'y repasser et de tourner en rond. Mais là s'arrête l'analogie. Lorsque la machine arrive à un cul-de-sac, en voyant que les sommets vers lesquels elle pourrait aller ont déjà été traversés, elle cherche une ouverture sur un des sommets par où elle est déjà passée. Elle va alors sauter directement du sommet cul-de-sac vers le sommet ouvert, ce que ne saurait faire un piéton. C'est la programmation récursive qui provoque ce retour arrière automatique (*back tracking* en anglais).

Prenons l'exemple du graphe de la *figure 10* (*à gauche*), avec le point de départ de l'exploration situé au sommet 0. Sur le dessin *au centre*, on voit que la machine s'enfonce dans le graphe jusqu'au sommet 6. Là elle tombe sur un cul-de-sac, car elle voit que tous les chemins disponibles mènent à des sommets déjà atteints. Alors elle saute directement sur le premier sommet en arrière où un chemin est ouvert, en attente d'être parcouru. Ici elle retourne au sommet 4 qui permet d'aller au sommet 7 qui n'était pas encore atteint. A son tour le sommet 7 est un cul-de-sac, et le retour arrière se fait jusqu'au sommet 0 d'où l'on peut aller au sommet 2. Finalement on se retrouve au sommet 0 de départ. De telles discontinuités dans le parcours, telle que la machine les pratique, n'est pas possible avec un piéton. Ce dernier devra faire de véritables marches arrière, comme indiqué sur la *figure 10 à droite*. Autrement dit, lorsque le piéton atteint le sommet 6, il fait demi-tour et revient sur ses pas jusqu'au sommet 4, d'où il voit que le sommet 7 peut être atteint. Puis il fait demi-tour sur ce sommet 7 et revient en arrière jusqu'au sommet 0, etc. Cette marche continue du piéton, calquée sur l'exploration en machine, suppose que le piéton voit les sommets à distance, car il est censé savoir s'ils ont déjà été traversés ou non. Par exemple, lorsque le piéton atteint le sommet 5, il doit voir qu'il est déjà passé en 0, et il n'emprunte pas la route qui y mène. Concrètement, cela demande que les sommets déjà atteints ne soient pas simplement marqués par un petit caillou, mais qu'on y plante un grand mât avec un drapeau, de façon à les voir de loin.[11] Cela suppose aussi que le chemin menant à un sommet déjà atteint soit en ligne droite, de façon à ne pas risquer de se tromper de sommet.

---

[9] Cela s'appelle aussi algorithme de Trémaux, du nom d'un pionnier en matière d'exploration de labyrinthe. C. P. Trémaux est un mathématicien français de la fin du 19è siècle, qui a été redécouvert récemment par des mathématiciens non-français, comme c'est souvent le cas.

[10] Précisons que le graphe est enregistré en machine en se donnant les listes d'adjacence des sommets. Par exemple le sommet 0 a pour voisins 1, 2, 5, 7. Nous avons choisi d'écrire ces listes de voisins dans l'ordre croissant, et la machine choisit le premier voisin disponible, par exemple à partir du sommet 0, elle commence à aller vers le sommet 1. La trajectoire d'exploration dépend de l'ordre dans lequel les voisins sont choisis.

[11] Pour les programmes correspondants, on pourra aussi se reporter à *Combien ? Mathématiques appliquées à l'informatique, tome 3 : graphes*. Ces programmes exploitent le fait que chaque sommet a trois états successifs : pas encore atteints, en cours d'exploration, et finis d'explorer.

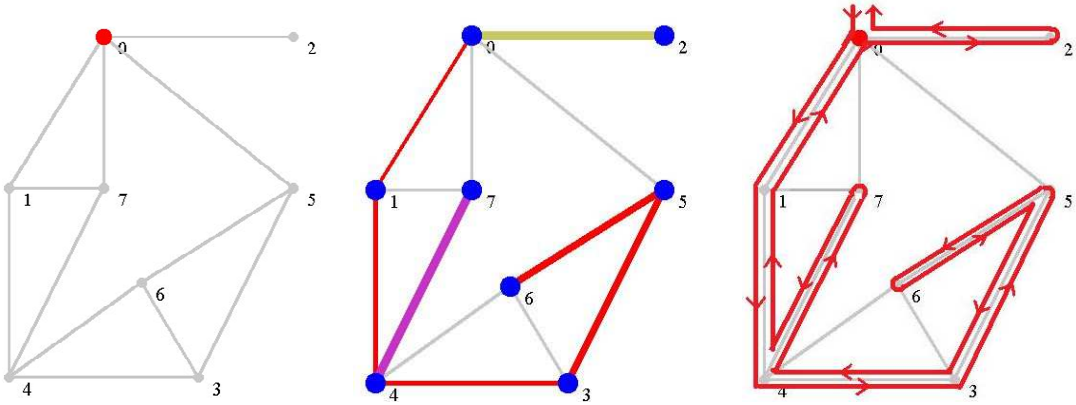


Figure 10 : *A gauche* le graphe initial avec le point de départ 0. *Au centre* l'exploration en profondeur telle que la réalise la machine, avec des discontinuités lors du parcours qui sont signalées par des changements de couleur. *A droite*, le cheminement correspondant d'un piéton, avec de véritables marches arrière

Remarquons que dans l'exemple de la *figure 10*, le trajet du piéton (qui voit loin) se fait en longeant toujours les murs à droite. Cela est dû à l'ordre dans lequel les voisins de chaque sommet ont été choisis. Si l'on change l'ordre des voisins, on aboutit aussi à un parcours du labyrinthe, comme par exemple sur la *figure 11*.

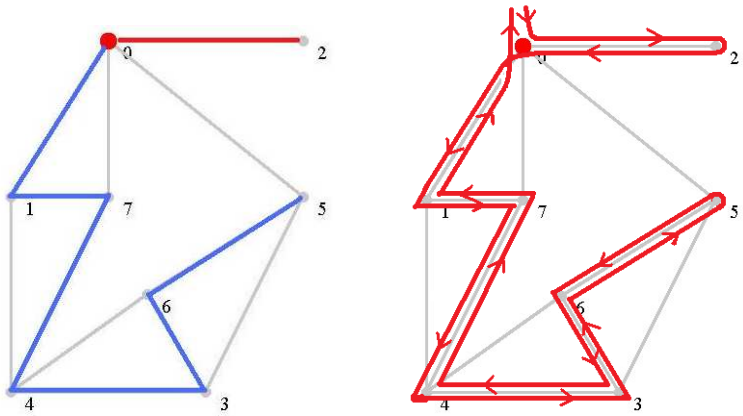


Figure 11 : *A gauche* l'exploration en profondeur, en commençant par aller de 0 à 2, avec un chemin qui a deux discontinuités, *à droite* l'exploration du piéton qui voit, avec ses deux retours arrière, de 2 à 0 et de 5 à 0

Maintenant passons au cas général, où le piéton ne voit rien à distance. Il devra aller jusqu'à un carrefour pour savoir s'il est déjà passé par là. Le cheminement du piéton qui voit, comme sur les *figures 10* et *11*, va devoir se prolonger sur les couloirs qui jusque là étaient laissés de côté. On obtient alors les cheminements de la *figure 12*. Si l'on respecte le fait de longer les murs à droite, on obtient le chemin de la *figure 12* à gauche. Notamment lorsque le piéton arrive au sommet 5, il se dirige sur le sommet 0, et quand il y arrive, il constate qu'il est déjà passé par là et fait demi-tour pour revenir vers le sommet 5. Plus précisément, alors que le piéton qui voit allait du sommet 6 au sommet 5 puis faisait demi-tour, le piéton qui ne voit pas à distance et qui arrive en 6, commence par emprunter la ou les routes libres à double sens, dans un sens puis dans l'autre, et c'est seulement après avoir pris ces routes qu'il revient de 6 à 5.

Remarquons que cela revient à remplacer le graphe initial avec ses arêtes par un graphe orienté où chaque arête est remplacée par deux arcs de sens opposé. L'exploration du labyrinthe nous conduit à parcourir chaque arc une fois et une seule. Cela signifie que le parcours est un cycle eulérien sur le

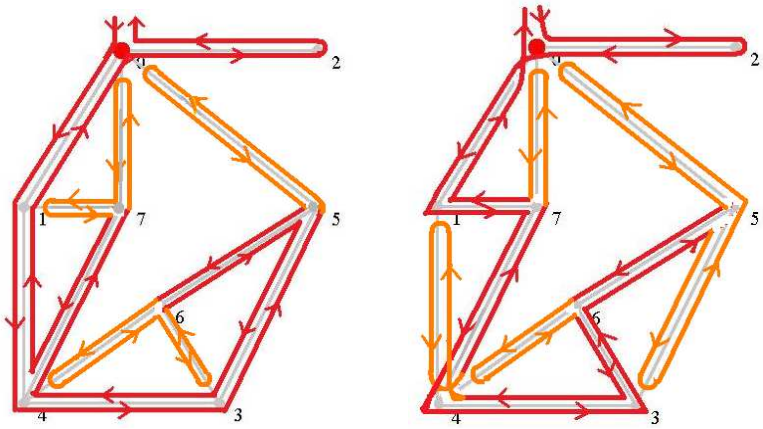


Figure 12 : le trajet complet du piéton dans le labyrinthe, en prolongeant les chemins des *figures 10* (*à gauche*) et *11* (*à droite*). Le chemin du piéton qui voit reste en rouge. Les couloirs supplémentaires parcourus à double sens sont indiqués en couleur ocre. Ils s'intercalent entre l'aller et le retour des chemins rouges

graphe connexe initial. Comme chaque sommet possède autant d'arcs entrants que d'arcs sortants, on sait [12] que le graphe possède des cycles eulériens. La méthode précédente permet d'en trouver un.

Pour explorer le labyrinthe, le piéton doit marquer chaque couloir qu'il emprunte en dessinant une flèche sur le mur, indiquant le sens de son parcours, au début et à la fin du couloir au moins.[13] De cette façon, il saura par la même occasion s'il est déjà passé en un carrefour ou pas. A la fin du cheminement les murs de chaque couloir seront marqués de deux flèches de sens opposé.

Passons aux règles du jeu qui permettent d'explorer le labyrinthe en totalité. Elles découlent de la construction du cheminement telle que nous l'avons faite.

1) Le piéton arrive pour la première fois en un carrefour, aucune flèche n'indiquant un passage antérieur. Dans ce cas, il prend n'importe quelle route qui s'ouvre devant lui (*figure 13 à gauche*). Et s'il n'en existe aucune ? Il fait alors demi-tour (*figure 13 à droite*).

2) Le piéton arrive en un carrefour où il est déjà passé, et le couloir qu'il vient d'emprunter n'a été parcouru qu'à sens unique. Dans ce cas il fait demi-tour, et reprend le même couloir dans l'autre sens (*figure 14*).

3) Le piéton arrive en un carrefour où il est déjà passé, et le couloir par lequel il arrive a été parcouru à double sens. Dans ce cas, il choisit en priorité un couloir qui n'a jamais été parcouru. S'il n'en existe pas, il prend un couloir déjà parcouru dans un sens (*figure 15*).

---

[12] Il s'agit d'un théorème attribué à L. Euler, même si ce phénomène était depuis longtemps exploité en Afrique, de l'Egypte au Congo, sur de nombreux motifs artistiques.

Comment démontrer ce théorème ? Partons d'un sommet, en empruntant un arc sortant. Chaque fois que l'on arrive à un sommet, par un arc entrant, on est sûr de pouvoir en sortir, par un arc sortant, puisque les arcs sortants sont aussi nombreux que les entrants. On ne reste donc jamais collé sur un sommet. Mais à cause du nombre fini de sommets, le chemin ne peut pas se poursuivre indéfiniment. Une seule possibilité se produit : le retour au point de départ, seul sommet dont on n'avait jusque là utilisé qu'un arc sortant. On vient d'obtenir un cycle sur le graphe. Mais ce n'est pas forcément un cycle eulérien, car il peut subsister des arcs non encore parcourus. Dans ce cas, on repart de l'origine d'un tel arc, et à partir de là comme précédemment on trouve un nouveau cycle. Quitte à répéter cela autant de fois qu'il faut, on est sûr d'obtenir des cycles qui parcourent tous les arcs du graphe. Il ne reste plus qu'à fusionner ces cycles pour avoir un cycle eulérien.

[13] On pourrait aussi utiliser un *fil d'Ariane*, sans jamais le rembobiner, ou encore en le rembobinant chaque fois qu'on a fait un aller-retour sur un sommet.

Figure 13 : *A gauche*, le piéton arrive à un carrefour pour la première fois, suivant la flèche rouge, en haut sur le dessin. Il repart vers l'un des chemins qui s'ouvrent devant lui (en couleur ocre). *A droite*, le piéton tombe sur un cul-de-sac, et il fait demi-tour.

Figure 14 : Le piéton arrive à un carrefour où il est déjà passé. Le couloir par lequel il arrive n'a pas été pour le moment parcouru dans l'autre sens. Le piéton fait demi-tour, et reprend le couloir dans l'autre sens

Figure 15 : Le piéton arrive à un carrefour où il est déjà passé, et le couloir par lequel il arrive a déjà été parcouru dans l'autre sens. S'il existe une ou plusieurs routes ouvertes à double sens, il en prend une (*à gauche*). Si aucune route n'est ouverte à double sens, il prend un couloir où il est déjà passé dans l'autre sens (*à droite*)

Dans ce contexte général, le fait de cheminer en longeant le mur à droite n'a pas d'intérêt précis.[14] Lorsque le piéton dispose de plusieurs possibilités, il peut en choisir une au hasard, à condition de bien marquer son choix par des flèches. On en déduit la programmation de l'exploration du labyrinthe.

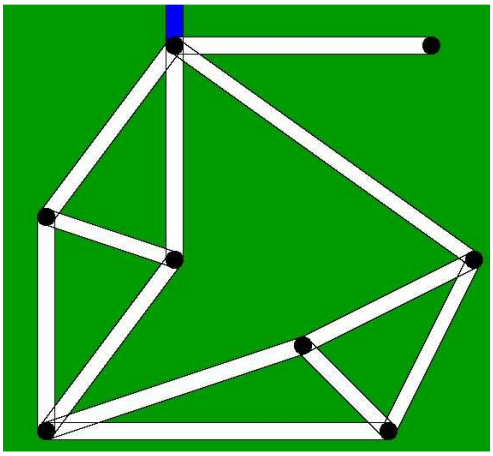
3.2. Programme

Le programme principal se contente d'appeler la fonction *graphe*() qui enregistre le graphe en machine et qui le dessine sur l'écran, puis la fonction *chemin*(0) à partir du point de départ choisi, ici 0.

```
graphe();
pred[0]=-1; chemin(0);  /* le fait de prendre comme prédécesseur du sommet 0 un sommet factice
                           – 1 est sans doute d'une prudence superflue */
```

[14] D'autre part, le fait de s'imposer d'avoir un mur à sa droite lors du cheminement suppose, lors de la programmation, que l'on ordonne de droite à gauche les couloirs qui s'ouvrent devant le piéton à chaque carrefour. Ce qui complique le programme si l'on veut l'appliquer à un graphe quelconque.

Précisons que la fonction *graphe*() compte le nombre des arcs (toujours à double sens) du graphe dans la variable *nbarcs*. Elle utilise aussi un tableau d'arcs *e*[*NS*][*NS*], *NS* étant le nombre des sommets, avec *e*[*i*][*j*] qui est nul s'il n'existe aucun arc de *i* vers *j*, et non nul sinon. Les coordonnées des vecteurs correspondant aux arcs sont placées dans (*vx*[*i*][*j*], *vy*[*i*][*j*]), et les coordonnées des sommets sont (*xe*[*i*],ye[*i*]), *i* étant le numéro de sommet concerné. Cette fonction *graphe*() étant liée au dessin du graphe choisi, elle doit être construite pas à pas, et son programme est assez long, même s'il ne présente pas de difficulté particulière. On aboutit pour notre graphe au dessin suivant, avec les sommets en noir, pour indiquer qu'aucun n'a pour le moment été traversé :



Passons à la fonction *chemin*(*i*) appelée sur le sommet *i* que l'on a atteint au cours de l'exploration, et qui se rappelle sur le sommet suivant, après le parcours d'un arc. Signalons que nous avons construit une fonction *linecircle*() qui dessine un disque (bleu dans le cas présent) se déplaçant sur la ligne, ce qui simule le cheminement du piéton. Cette fonction est calquée sur la fonction *line*(), avec les points remplacés par des disques. Chaque fois qu'un arc est parcouru, on diminue *nbarcs* de 1, et la fonction se rappelle tant que *nbarcs* est différent de 0. On applique ensuite les règles du jeu précédemment données. On aboutit à cette fonction :

```
void chemin(int i)
{  int j,flag=0;  int h,nbaretes,ii[NS],jj[NS];
   if (nbarcs!=0)
     {  if (dejavu[i]==1 && e[i][pred[i]]!=0) /* cas où l'on tombe sur sommet déjà atteint, avec le couloir par
                                                 lequel on arrive qui n'a été parcouru que dans un sens. On
                                                 fait demi-tour, et on retourne où on était */
        {  e[i][pred[i]]=0; pred[pred[i]]=i;  /* l'arc de retour est mis à 0 */
           linecircle(xe[i]+vx[i][pred[i]],ye[i]+vy[i][pred[i]],  /* dessins */
                      xe[pred[i]]+vx[i][pred[i]],ye[pred[i]]+vy[i][pred[i]],blue,2); SDL_Flip(screen);
           linewithwidth(xe[i]+vx[i][pred[i]],ye[i]+vy[i][pred[i]],
                         xe[pred[i]]+vx[i][pred[i]],ye[pred[i]]+vy[i][pred[i]],2,red);
           arrow(xe[i]+vx[i][pred[i]],ye[i]+vy[i][pred[i]],
                 xe[pred[i]]+vx[i][pred[i]],ye[pred[i]]+vy[i][pred[i]],red);SDL_Flip(screen);
           nbarcs--;
           chemin(pred[i]); /* la fonction se rappelle sur le sommet qui était le prédécesseur de i */
        }
      else if ( dejavu[i]==1)  /* cas où l'on tombe sur sommet déjà atteint, avec le couloir par
                                  lequel on arrive qui a été parcouru dans les deux sens */

        { nbaretes=0;  /* on va chercher un couloir ouvert à double sens */
          for(j=0;j<NS;j++)
          if (e[i][j]!=0 && e[j][i]!=0)
             {ii[nbaretes]=i; jj[nbaretes++]=j; }
          if (nbaretes>0) /* cas où il existe des couloirs ouverts à double sens. On en prend un au hasard */
             {h=rand()%nbaretes;
              i=ii[h];j=jj[h];
              pred[j]=i;  e[i][j]=0;
```

```
                linecircle(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],blue,2);SDL_Flip(screen);
                arrow(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],red);SDL_Flip(screen);
                linewithwidth(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],2,red);
                nbarcs--;
                chemin(j);
               }
            else if (nbaretes==0)   /* aucun couloir ouvert dans les deux sens. On choisit un couloir déjà
                                       parcouru dans l'autre sens */
              { nbaretes1=0;
                for(j=0;j<NS;j++)
                if (e[i][j]!=0 ) {ii[nbaretes1]=i; jj[nbaretes1++]=j; }
                h=rand()%nbaretes1;
                i=ii[h];j=jj[h];
                pred[j]=i;  e[i][j]=0;
               linecircle(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],blue,2);SDL_Flip(screen);
               arrow(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],red);SDL_Flip(screen);
               linewithwidth(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],2,red);
               nbarcs--;
               chemin(j);
              }
          }
        else if   (dejavu[i]==0  ) /* cas où le sommet i est atteint pour la première fois */
          { dejavu[i]=1; filldisc(xe[i],ye[i],10,red); /* ce sommet est mis en rouge */
            flag=0;
            nbaretes=0;
             for(j=0;j<NS;j++)
             if (e[i][j]!=0 && j!=pred[i])  /* on cherche au hasard un couloir, sauf celui d'où l'on vient */
               {ii[nbaretes]=i; jj[nbaretes++]=j;  flag=1;}
             if (nbaretes>0)  /* s'il y a plusieurs couloirs, on en prend un au hasard */
               { h=rand()%nbaretes;
                 i=ii[h];j=jj[h];
                 pred[j]=i;  e[i][j]=0;
                 linecircle(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],blue,2);SDL_Flip(screen);
                 linewithwidth(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],2,red);
                arrow(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],red);SDL_Flip(screen);
               nbarcs--;
               chemin(j);
               }
             else if (flag==0)  /* cas où l'on est tombé sur un cul-de-sac */
               { for(j=0;j<NS;j++)
                 if (e[i][j]!=0 ) break;
                 pred[j]=i;  e[i][j]=0;
                 linecircle(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],blue,2);SDL_Flip(screen);
                 linewithwidth(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],2,red);
                 arrow(xe[i]+vx[i][j],ye[i]+vy[i][j],xe[j]+vx[i][j],ye[j]+vy[i][j],red);SDL_Flip(screen);
                 nbarcs--;
                 chemin(j);
               }
           }
      }
}
```

On verra sur la *figure 16* comment se fait l'exploration du labyrinthe, avec certaines routes prises au hasard quand plusieurs possibilités existent.

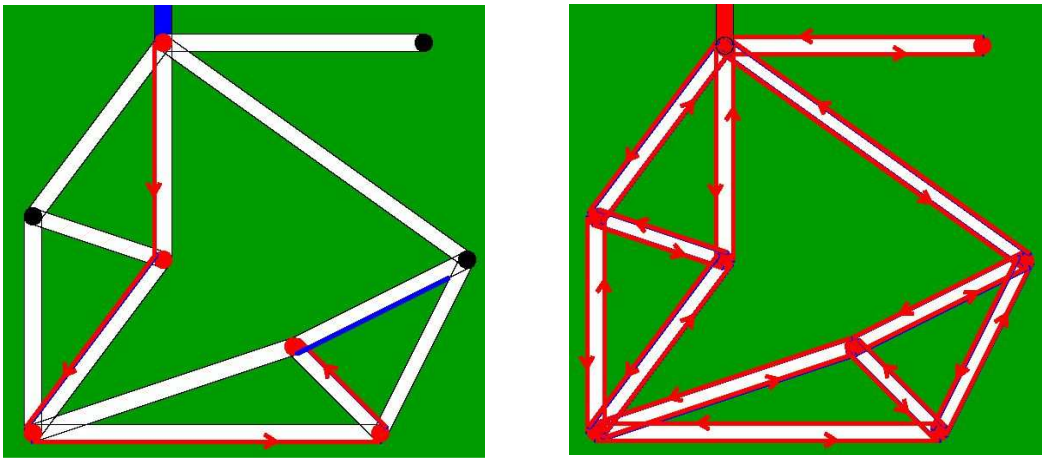


Figure 16: *A gauche* le labyrinthe en cours d'exploration, les sommets atteints passant de noir à rouge, et l'arc en cours de parcours est dessiné en bleu au fil de son parcours, avant de passer en rouge une fois son parcours terminé ; *à droite* l'exploration terminée, le piéton étant revenu au point de départ 0 après être passé partout

3.3. Cas particulier : labyrinthe à couloirs dans deux directions orthogonales

Dans ce contexte, où tous les couloirs sont tous parallèles ou à angle droit, avec la même largeur appelée *pas*[15] partout, l'algorithme où l'on suit toujours les murs du même côté s'applique plus aisément. On va choisir de garder les murs sur sa gauche. Commençons par dessiner un labyrinthe, comme celui de la *figure 17*, ce que fait la fonction *laby()*, longue à écrire mais sans aucune difficulté, aussi ne la donnons-nous pas ici.

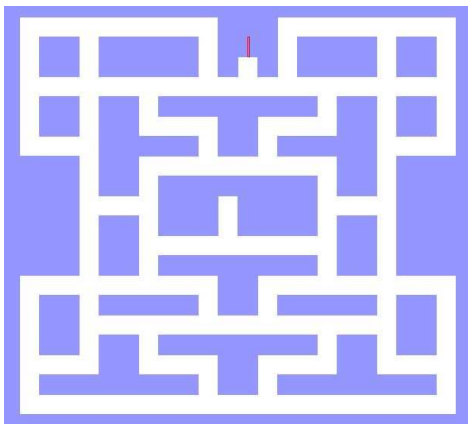


Figure 17 : Un exemple de labyrinthe, avec l'entrée indiquée par un trait rouge, et les couloirs en blanc

L'algorithme du cheminement du piéton va être purement graphique. Les couloirs sont découpés en carrés successifs de côté *pas*, le cheminement se faisant du centre d'un carré au centre du carré suivant. Ces centres vont avoir trois couleurs possibles : blanc quand le piéton ne les a pas encore atteints, rouge quand le piéton passe dessus pour la première fois, et vert quand il passe dessus la seconde fois. Ils jouent un peu le rôle des petits cailloux du Petit Poucet. D'autre part, chaque fois qu'un carré d'un couloir est atteint, le chemin va être dessiné le long du mur de gauche par un rectangle rouge. A la fin du parcours, le chemin du piéton est dessiné sous forme d'un large trait continu longeant les murs à gauche, avec en plus des demi-tours de temps à autre.

[15] On aura intérêt à prendre *pas* pair, afin que les programmes graphiques qui suivent ne fassent pas d'arrondi lorsque l'on divise *pas* par 2, ce qui peut être une source d'erreur.

Le piéton se comporte comme un mobile dont on connaît non seulement la position (*x, y*) à chaque pas, mais aussi la direction *dir*, qui vaut 0, 1, 2 ou 3 selon qu'elle est dirigée vers l'est, le nord, l'ouest ou le sud. Au départ, le piéton est au point (*xd, yd*) avec les direction *dir* = 3, puisqu'il doit aller vers le sud, ce qui donne comme conditions initiales, dans le contexte de notre labyrinthe :

```
xd=xorig+pas*11.5; yd=yorig-pas*17.5; dir=3; x=xd;  y=yd; /* (x, y) est le point courant */
```

Puis on lance une boucle infinie *for(;;)* qui s'arrêtera lorsque le point (*x, y*) se retrouvera au point de départ après avoir parcouru tout le labyrinthe à double sens. A chaque étape de la boucle, on commence par compter le nombre de couloirs possibles autour du point où l'on est (*oldx = x, oldy = y, olddir = dir*) où l'on est, sans compter le couloir d'où l'on vient, de direction *interdit*, ni les couloirs déjà parcourus à double sens. Ce nombre, placé dans *cumul*, vaut 0, 1, 2 ou 3. Supposons que le piéton regarde de droite à gauche. Le comptage se fait alors en tournant dans l'ordre inverse des aiguilles d'une montre à partir de *interdit*, sur les voisins du point, et l'on ne compte pas les cas où l'on tombe dans un mur (point de couleur bleue) ou sur un point vert, signifiant que le couloir correspondant a déjà été parcouru à double sens. Ce qui donne comme début de la boucle *for*(;;) :

```
for(;;)
  { oldx=x; oldy=y; olddir=dir;  interdit=(olddir+2)%4;
    for(i=0;i<4;i++) compteur[i]=0;
    for(i=(interdit+1)%4;i<(interdit+1)%4+3;i++)
      { if (i%4==0 && getpixel(x+pas,y)!=blue && getpixel(x+pas,y)!=vert) compteur[0]=1;
        if (i%4==1 && getpixel(x,y-pas)!=blue && getpixel(x,y-pas)!=vert) compteur[1]=1;
        if (i%4==2 && getpixel(x-pas,y)!=blue && getpixel(x-pas,y)!=vert) compteur[2]=1;
        if (i%4==3 && getpixel(x,y+pas)!=blue && getpixel(x,y+pas)!=vert) compteur[3]=1;
      }
    cumul=0;
    for(i=0;i<4;i++) cumul+=compteur[i];
```

A ce stade, trois cas se présentent :

1) La variable *cumul* vaut 0. Cela signifie que le piéton est arrivé à un cul-de-sac. il fait donc demi-tour, sans changer de position, et le point où il se trouve est mis en vert. Ce qui donne comme suite du programme, toujours dans la boucle *for* :

```
    if (cumul==0) {      dir=interdit; uturn(x,y,dir); filldisc(x,y,2,vert); }
```

2) La variable *cumul* est supérieure à 1. Deux ou trois couloirs sont ouverts. Cela signifie que l'on est à un carrefour. On commence par colorier le centre du carrefour avec un carré bleu (*bleu2* dans le programme). Ainsi lors de l'exploration du profondeur, lors du premier passage sur les carrefours du labyrinthe, ceux-ci seront tous coloriés en bleu. Puis en tournant de droite à gauche, on choisit le couloir le plus à gauche, de direction *dir*. Si cette direction *dir* est la même que celle, *olddir*, que le mobile avait, on va tout droit jusqu'au point suivant. Si l'on passe de *olddir* à *dir* en tournant de +90°, on tourne à gauche. On colorie le point où l'on est soit en rouge (*rouge2* dans le programme) soit en vert selon que l'on y est pour la première ou la deuxième fois. Enfin on détermine les coordonnées du point suivant, soit (*x, y*) en lui donnant aussi la direction *dir*. D'où la suite du programme dans la boucle :

```
    else if (cumul>1)
      { rectangle(x-pas/10,y-pas/10,x+pas/10,y+pas/10,blue);   floodfill(x,y,bleu2,blue);
        i=(interdit+3)%4;
        while(compteur[i]==0) i=(i+3)%4;
        dir=i;
        if (dir==olddir) {toutdroit(x,y,dir);}
         else if(dir==(olddir+1)%4)  agauche(x,y,dir);
         else if (olddir==(dir+1)%4) adroite(x,y,dir);
         if (dir==0) x+=pas;  else if (dir==1) y-=pas; else if (dir==2) x-=pas;  else if (dir==3) y+=pas;
      }
```

3) La variable *cumul* vaut 1. On détermine la seule direction à prendre, soit *dir*, ainsi que le point voisin (*newx*, *newy*). Mais on ne va pas forcément vers ce point. Deux cas se présentent :

- Si on est pour la première fois en (*x, y)* et que le point (*newx*, *newy*) est un carrefour où l'on est déjà passé, on fait demi-tour.

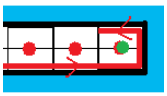
- Sinon, on avance vers le point suivant que l'on note à son tour (*x*, *y*), avec la direction *dir*.

On en déduit la fin de la boucle *for* :

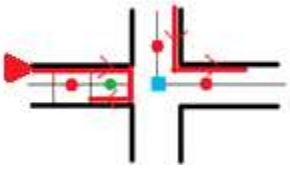
```
else if (cumul==1)
  { i=0; while (compteur[i]==0) i++;
    dir=i;
    if (getpixel(x,y)==white)   filldisc(x,y,2,rouge2);  else       filldisc(x,y,2,vert);
    if (dir==0) {newx=x+pas;newy=y;}    else if (dir==1) {newy=y-pas;newx=x;}
    else if (dir==2) {newx=x-pas;newy=y;}   else if (dir==3) {newy=y+pas;newx=x;}
    if (getpixel(newx,newy)==bleu2 && getpixel(x,y)==rouge2)
      { dir=interdit;  filldisc(x,y,2,vert); uturn(x,y,dir);
        if (dir==0) x+=pas;  else if (dir==1) y-=pas;   else if (dir==2) x-=pas; else if (dir==3) y+=pas;
      }
    else
      { if (dir==olddir) {toutdroit(x,y,dir);   }
        else if(dir==(olddir+1)%4)  square(x,y,dir);
        else if (olddir==(dir+1)%4) rightangle(x,y,dir);
        if (dir==0) x+=pas;  else if (dir==1) y-=pas;   else if (dir==2) x-=pas;   else if (dir==3) y+=pas;
      }
   }
  SDL_Flip(screen);SDL_Delay(50); /* affichage sur l'écran */
  if (x==xd && y==yd) { toutdroit(xd,yd,1); break;}  /* test d'arrêt, fin du cheminement */
}
```

Ce faisant, on a retrouvé les règles du jeu données au paragraphe 3.1., légèrement simplifiées. A la différence du cas général, le fait de choisir de suivre un mur à sa gauche, dans un contexte où le nombre de couloirs ne dépasse pas trois, n'impose pas de marquer les murs avec des flèches, mais avec des points (blancs, rouges ou verts).

Voici quelques exemples de cheminement, avec l'adjonction des fonctions auxiliaires qui dessinent un morceau du chemin dans le carré concerné du couloir.



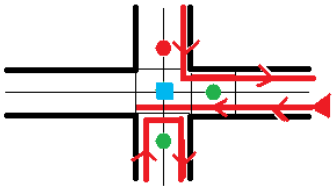
Arrivée dans un cul-de-sac, avec *cumul* = 0, *dir* = 0, et la direction devient *dir* = *interdit* = 2. On dessine le demi-tour, grâce à la fonction *demitour*(), qui fabrique une forme en U grâce à trois rectangles rouges.



Ici, on arrive pour la première fois au bout d'un couloir (point rouge) et cumul = 1, avec un carrefour voisin où l'on est déjà passé (carré bleu). On fait demi-tour et le point rouge est mis en vert.

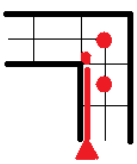
```
void demitour(int x, int y,int d)
{
if (d==0)
  { rectangle(x-pas/4,y-pas/4,x+pas/2,y-pas/2,red);  floodfill(x+pas/8,y-3*pas/8,red,red);
```

```
   rectangle(x-pas/4,y+pas/4,x+pas/2,y+pas/2,red);   floodfill(x+pas/8,y+3*pas/8,red,red);
   rectangle(x-pas/2,y-pas/2,x-pas/4,y+pas/2,red);   floodfill(x-3*pas/8,y,red,red);
  }
else if (d==2)
  { rectangle(x+pas/4,y-pas/4,x-pas/2,y-pas/2,red);  floodfill(x-pas/8,y-3*pas/8,red,red);
   rectangle(x+pas/4,y+pas/4,x-pas/2,y+pas/2,red);  floodfill(x-pas/8,y+3*pas/8,red,red);
   rectangle(x+pas/2,y-pas/2,x+pas/4,y+pas/2,red);  floodfill(x+3*pas/8,y,red,red);
  }
else if (d==1)
  { rectangle(x-pas/4,y+pas/4,x-pas/2,y-pas/2,red);  floodfill(x-3*pas/8,y-pas/8,red,red);
   rectangle(x+pas/4,y+pas/4,x+pas/2,y-pas/2,red);   floodfill(x+3*pas/8,y-pas/8,red,red);
   rectangle(x-pas/2,y+pas/2,x+pas/2,y+pas/4,red);  floodfill(x,y+3*pas/8,red,red);
  }
else if (d==3)
  { rectangle(x-pas/4,y-pas/4,x-pas/2,y+pas/2,red);   floodfill(x-3*pas/8,y+pas/8,red,red);
   rectangle(x+pas/4,y-pas/4,x+pas/2,y+pas/2,red);  floodfill(x+3*pas/8,y+pas/8,red,red);
   rectangle(x-pas/2,y-pas/2,x+pas/2,y-pas/4,red);  floodfill(x,y-3*pas/8,red,red);
  }
}
```



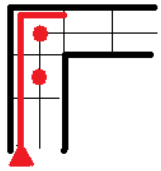
On arrive en un carrefour pour la deuxième fois (carré bleu), avec deux couloirs ouverts : *cumul* = 2. On choisit la direction la plus à gauche possible, ici c'est tout droit, avec l'appoint de la fonction *toutdroit*(), qui ici avec *dir* = 2 dessine un rectangle rouge horizontal en bas, indiquant le chemin.

```
void toutdroit(int x,int y,int d)
{
if (d==0)  { rectangle(x-pas/2,y-pas/2, x+pas/2,y-pas/4,red);
             floodfill(x,y-3*pas/8,red,red);
              fleche(x-pas/2, y-3*pas/8, x+pas/2,y-3*pas/8,red);
             }
else if (d==2)  { rectangle(x-pas/2,y+pas/2, x+pas/2,y+pas/4,red);
                 floodfill(x,y+3*pas/8,red,red);
                  fleche(x+pas/2, y+3*pas/8, x-pas/2,y+3*pas/8,red);
                 }
else if (d==1) { rectangle(x-pas/2,y-pas/2, x-pas/4,y+pas/2,red);
                   floodfill(x-3*pas/8,y,red,red);
                   fleche(x-3*pas/8, y+pas/2, x-3*pas/8,y-pas/2,red);
                 }
else if (d==3) { rectangle(x+pas/2,y-pas/2, x+pas/4,y+pas/2,red);
                 floodfill(x+3*pas/8,y,red,red);
                 fleche(x+3*pas/8, y+pas/2, x+3*pas/8,y-pas/2,red);
                 }
}
```



On arrive à un tournant à gauche, avec *cumul* = 1. Dans le carré concerné, avec la direction passant de 1 à 2, on dessine un petit carré rouge dans le coin en bas à gauche, pour assurer la continuité du chemin, grâce à la fonction *agauche*().

```
void agauche(int x,int y, int d)
{
if (d==0) { rectangle(x+pas/4,y-pas/4,x+pas/2,y-pas/2,red); floodfill(x+3*pas/8,y-3*pas/8,red,red); }
else if (d==1) { rectangle(x-pas/4,y-pas/4,x-pas/2,y-pas/2,red);  floodfill(x-3*pas/8,y-3*pas/8,red,red);  }
else if (d==2) { rectangle(x-pas/4,y+pas/4,x-pas/2,y+pas/2,red); floodfill(x-3*pas/8,y+3*pas/8,red,red); }
else if (d==3) { rectangle(x+pas/4,y+pas/4,x+pas/2,y+pas/2,red); floodfill(x+3*pas/8,y+3*pas/8,red,red); }
}
```



On arrive à un tournant à droite, avec cumul = 1. Dans le carré concerné, où la direction passe de 1 à 0, on dessine des rectangles rouges bordant le côté de gauche et le côté du haut, pour montrer le chemin suivi, grâce à la fonction *adroite*()

```
void adroite(int x,int y, int d)
{
if (d==0)
  { rectangle(x-pas/4,y-pas/4,x+pas/2,y-pas/2,red);  floodfill(x+pas/8,y-3*pas/8,red,red);
    rectangle(x-pas/4,y-pas/4,x-pas/2,y+pas/2,red);  floodfill(x-3*pas/8,y+pas/8,red,red);
    rectangle(x-pas/4,y-pas/4,x-pas/2,y-pas/2,red);  floodfill(x-3*pas/8,y-3*pas/8,red,red);
  }
else if (d==1)
  { rectangle(x-pas/4,y+pas/4,x-pas/2,y-pas/2,red);    floodfill(x-3*pas/8,y+pas/8,red,red);
    rectangle(x-pas/4,y+pas/4,x+pas/2,y+pas/2,red);   floodfill(x+pas/8,y+3*pas/8,red,red);
    rectangle(x-pas/4,y+pas/4,x-pas/2,y+pas/2,red);  floodfill(x-3*pas/8,y+3*pas/8,red,red);
  }
else if (d==2)
  { rectangle(x+pas/4,y+pas/4,x+pas/2,y-pas/2,red);  floodfill(x+3*pas/8,y-pas/8,red,red);
   rectangle(x+pas/4,y+pas/4,x-pas/2,y+pas/2,red);   floodfill(x-pas/8,y+3*pas/8,red,red);
    rectangle(x+pas/4,y+pas/4,x+pas/2,y+pas/2,red);  floodfill(x+3*pas/8,y+3*pas/8,red,red);
  }
else if (d==3)
  { rectangle(x+pas/4,y-pas/4,x-pas/2,y-pas/2,red);  floodfill(x-pas/8,y-3*pas/8,red,red);
    rectangle(x+pas/4,y-pas/4,x+pas/2,y+pas/2,red);  floodfill(x+3*pas/8,y+pas/8,red,red);
    rectangle(x+pas/4,y-pas/4,x+pas/2,y-pas/2,red);  floodfill(x+3*pas/8,y-3*pas/8,red,red);
  }
}
```

Les résultats obtenus sont indiqués sur la *figure 18*.

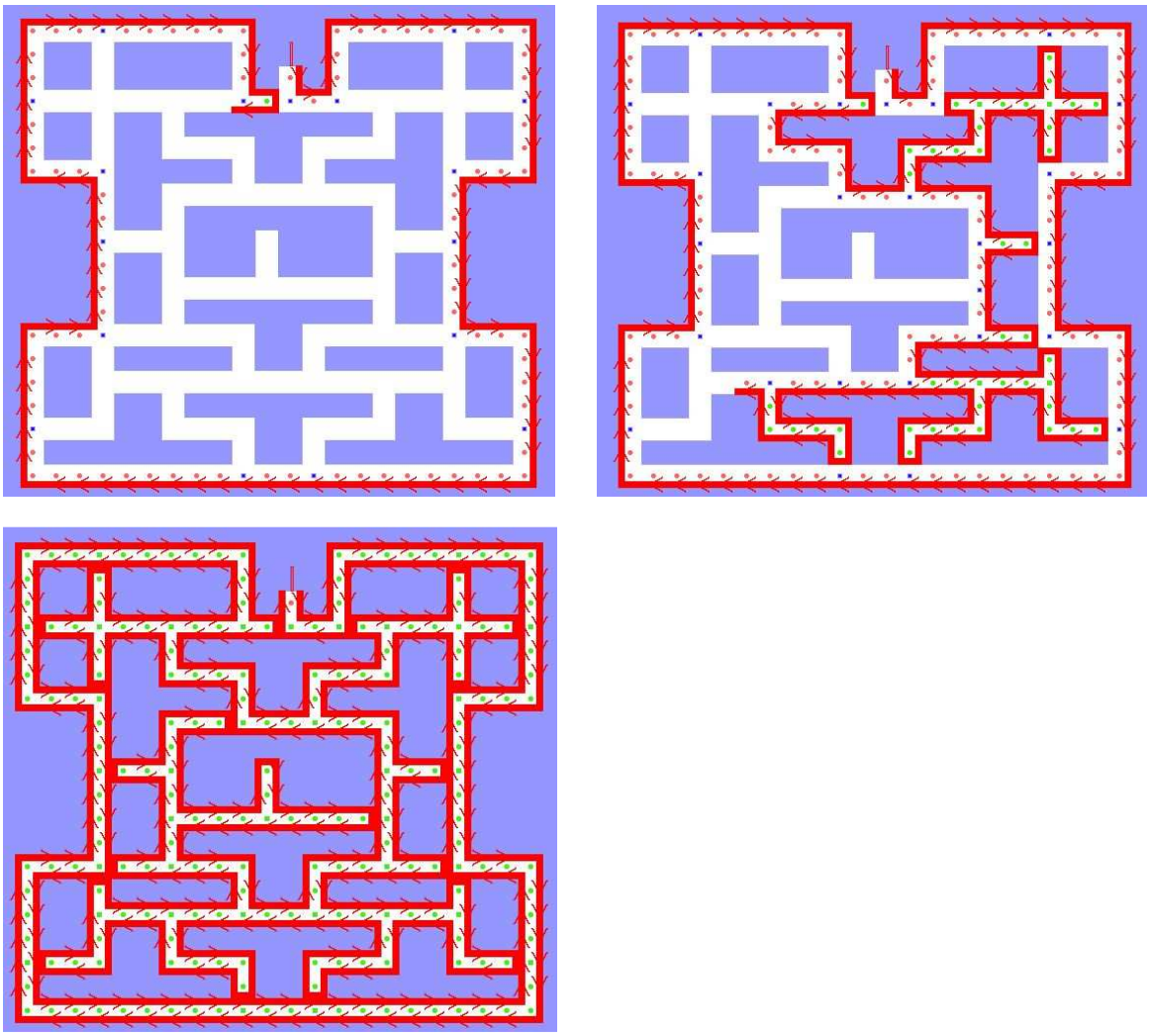


Figure 18 : *En haut*, le cheminement en cours, *en bas*, le chemin jusqu'au bout