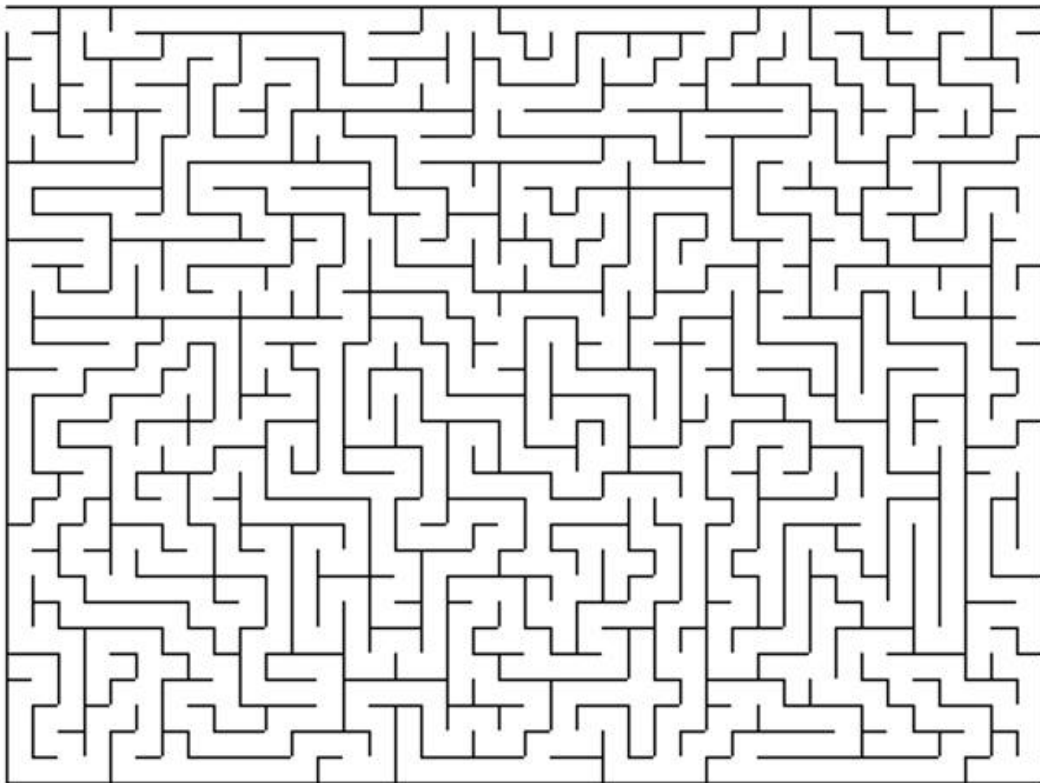


# **TIPE** : Programme de génération aléatoire de labyrinthes.



Labyrinthe créé par notre programme.

Elèves :  
Benoit De Dinechin  
Armand Gonthier  
Abel Samot

## **Sommaire :**

### **I/ Les labyrinthes**

#### **1) Qu'est-ce qu'un labyrinthe ?**

- a. Définition
- b. Présentation des différents types de labyrinthes
- c. Les labyrinthes Manieristes
- d. Comment trouver la solution d'un labyrinthe ?

### **II/ La programmation d'un labyrinthe**

#### **1) La structure des labyrinthes en informatique**

#### **2) Les différentes méthodes pour générer un labyrinthe**

- 1.1) [Première méthode étudiée](#)
- 1.2) [Deuxième méthode étudiée](#)
- 1.3) [Troisième méthode étudiée](#)
- 1.4) [Quatrième méthode étudiée](#)
- 1.5) [Cinquième méthode étudiée](#)

#### **3) La méthode choisie : Explication du principe et du programme**

- a. Explication rapide du principe
- b. Explication du programme.
- c. Analyse statistique du temps nécessaire pour résoudre les labyrinthes générés par le programme.

## **Intro :**

*Pour ce TIPE nous avons voulu créer un programme de génération aléatoire de Labyrinthe. C'est à dire un programme capable de générer, seule, des milliers de labyrinthes différents les uns des autres.*

*La notion de labyrinthe peut au premier abord sembler très simple. En effet dès tout petit on encourage les enfants à résoudre des labyrinthes pour entrainer leur logique. De plus si on demandait à quelqu'un de tracer un labyrinthe celui-ci y arriverait sans problème. Mais arriverait-il à créer un labyrinthe assez difficile pour ne pas être résolu du premier coup ?*

*C'est en regardant le film "Inception" de Christopher Nolan que nous nous sommes interrogés à ce sujet. En effet au début du film le héros demande à une jeune architecte de dessiner en 2 minutes un labyrinthe prenant 1 minute à résoudre. Nous avons nous même fait le test et il s'avère très difficile à réaliser quand on n'y connaît rien ou peu. On peut donc se demander si il existe une technique spécifique pour tracer des labyrinthes d'une certaine difficulté ou si il s'agit seulement de logique et de rapidité. L'ordinateur étant bien plus rapide que l'humain et pouvant de ce fait tester des milliers de possibilités en moins d'une seconde, nous nous sommes donc orienté vers le tracé de labyrinthe par ordinateur.*

*Cependant les ordinateurs que nous utilisons ne peuvent quant à eux pas faire preuve de logique afin de générer un labyrinthe qui doit non seulement comporter une solution mais qui doit aussi prendre à un humain un certain temps à résoudre. Il va donc falloir que l'ordinateur fasse de nombreux test avant de finir par afficher un labyrinthe convenable. C'est là que se trouve la difficulté, nous tacherons donc au cours de ce dossier de répondre à une problématique bien précise :*

**Comment peut on créer un programme capable de générer aléatoirement des milliers de labyrinthes différents, ayant tous une solution ? Ces labyrinthes doivent mètre en moyenne plus d'une minute à résoudre.**

*Nous commencerons par parler des labyrinthes en général : les différents types de labyrinthes qui existent et les méthodes de résolution.*

*Puis nous présenterons les différents algorithmes capables de répondre à la problématique ci-dessus et détaillerons leurs avantages et leurs inconvénients.*

*Nous finirons par présenter l'algorithme choisi ici en le détaillant ainsi qu'en expliquant comment nous l'avons adapté au langage informatique C.*

# I/ Les labyrinthes

## 1) Qu'est-ce qu'un labyrinthe ?

### a. Définition

#### Labyrinthe:

(du grec *laburinthos*, palais des haches)

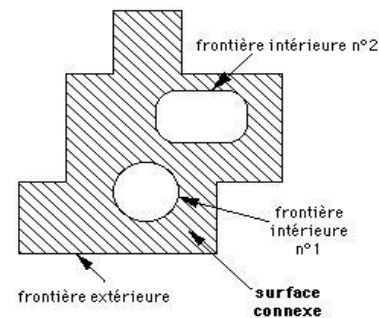
Edifice tracé muni ou non d'embranchements, d'impasses et de fausses pistes afin qu'on en trouve difficilement l'issue.

Cet édifice peut être sous forme de bâtiments, de plantes ou de simples traits sur une feuille de papier. C'est le tracé qui compte.

### b. Présentation des différents types de labyrinthes

Un labyrinthe est une surface connexe, c'est à dire qu'il est fermé par une unique frontière extérieure (comme le schéma ci-contre).

Il peut par contre contenir plusieurs frontières intérieures (dans le schéma on peut en voir deux). On appelle « îlot » les zones formées par ces frontières intérieures.

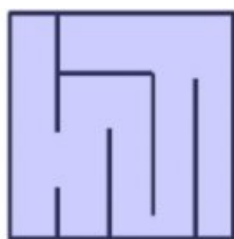


Les surfaces Connexes avec îlot et sans îlot sont dites topologiquement différentes (c'est à dire que ce sont des réseaux de communications différents vérifiant des propriétés différentes).

Cette différence explique l'existence de deux types de labyrinthes : Les labyrinthes parfaits et les labyrinthes imparfaits.

Les labyrinthes parfaits sont des surfaces connexes sans îlot, c'est à dire que tous les murs sont reliés les uns aux autres. Il existe un chemin unique.

Les labyrinthes imparfaits sont des surfaces connexes avec îlots. Ils peuvent posséder plusieurs chemins.



Labyrinthe *parfait*



Labyrinthes *imparfaits*

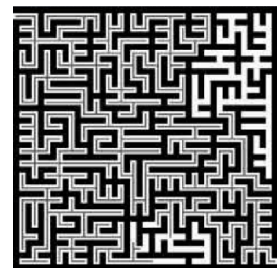
Nous nous intéresserons ici aux Labyrinthes parfaits, afin que le labyrinthe généré ne soit pas trop facile à résoudre.  
Parmis ces deux types de labyrinthes il existe des labyrinthes à caractéristiques différentes.

- Les labyrinthes “unicursal” sont ceux présents dans la mythologie grecque (labyrinthe du minotaure dont on parlera plus loin).  
Le parcours se fait de l’extérieur au centre et ne compte pas d’impasses ni d’embranchements.  
Ce n’est pas ce type de labyrinthe qui nous intéressera ici car il ne nécessite pas forcément de logique pour être résolu, seulement de la persévérance.



*Labyrinthe  
unicursal*

- Les labyrinthes en “Rhizome” qualifiés aussi “d’hermétiques” sont des réseaux infinis qui n’ont ni entrée ni sortie.  
Ils comportent des croisements et des impasses mais la technique du fil d’Ariane y est inutile.  
De plus, chaque route peut être la bonne tant que l’on va vers le côté du labyrinthe désiré.  
C’est un type de labyrinthe dont la résolution repose sur du hasard, c’est pour cela qu’il ne nous intéressera pas par la suite.



*Labyrinthe  
en Rhizome*

- Finalement, les labyrinthes qui nous intéresseront sont les labyrinthes maniéristes. Ils répondent aux conditions qui sont de contenir une unique solution et de poser un minimum de difficulté à celui qui souhaite les résoudre. Ils présentent un grand nombre de voies mais toutes exceptée une, mènent à un cul-de-sac.

### c. Les labyrinthes Manieristes

Les labyrinthes manieristes parfaits peuvent être représentés sous la forme d'arbres binaires. Ce sont des structures de données très particulières qui ont été créées dans le but de représenter des données identiques (dans notre cas, les cases du labyrinthe et les différents chemins de celui-ci).

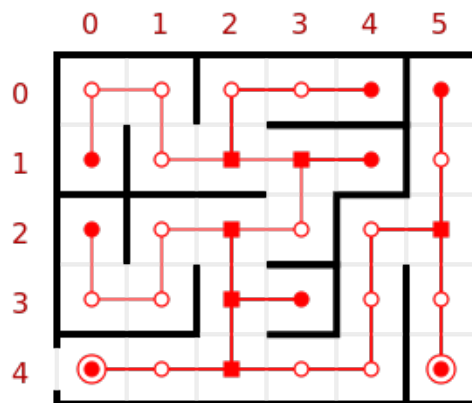
L'entrée du labyrinthe sera alors représentée par la racine de l'arbre (comme on peut voir ci-dessous).

Les branches de cet arbre représentent toutes les voies du labyrinthe. On peut voir que toutes ces branches exceptée une, mènent à un cul de sac.

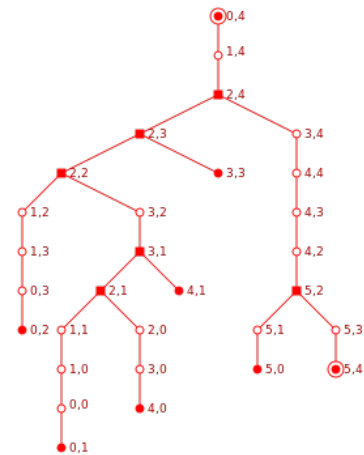
Il est interdit de passer d'une branche à l'autre autrement qu'en revenant au noeud qui relie les différentes branches.

Cette structure sera par la suite très intéressante pour notre algorithme. En effet, le fait de revenir en arrière après un cul de sac sera une des bases de celui-ci.

Il suffira ensuite de conférer un caractère aléatoire au tout pour déjà avoir une base d'algorithme.



*Labyrinthe manieriste parfait et son arbre associé.*



*Représentation en arbre du même labyrinthe.*

Finalement, on peut voir avec ces arbres que les labyrinthes parfaits manieristes sont structurés par cellules.

#### d. Comment trouver la solution d'un labyrinthe ?

Pour un labyrinthe parfait il existe une technique très simple pour trouver à coup sûr la solution. Il suffit de suivre toujours le même mur (qui peut se trouver soit à droite soit à gauche de soi) sans lâcher celui-ci. On atteindra forcément l'arrivée.

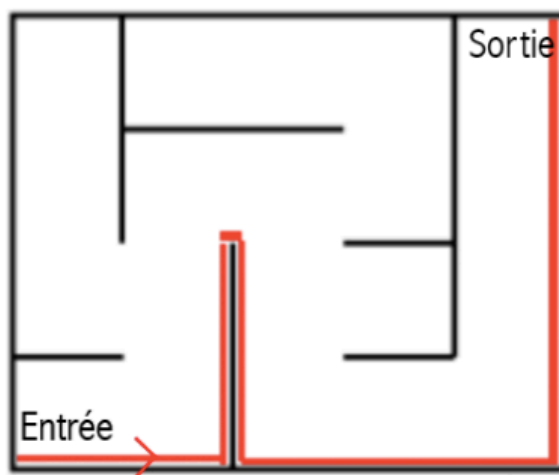
Les labyrinthes non parfaits sont les seuls pour lesquels cette technique peut ne pas fonctionner (exemple schéma ci contre).

En effet, si quelqu'un n'arrive pas à sortir d'un labyrinthe en utilisant cette technique, c'est qu'il tourne en rond.

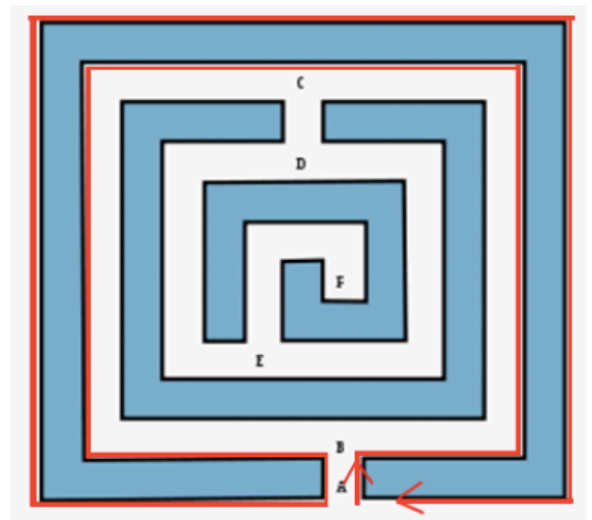
La seule possibilité pour que l'on ne tourne pas en rond est la présence d'îlots comme il y en a dans les labyrinthes non parfaits.

Or il est évident qu'il est impossible d'atteindre un tel îlot en suivant le même mur.

On pourrait donc se contenter de cette méthode.



Labyrinthe « parfait »



Labyrinthe « imparfaits »

#### Légende :

<sup>e</sup> En rouge on a le chemin pris si l'on suit un seul mur.

p

C

Cependant le principal problème de cette méthode est qu'elle peut s'avérer très longue. Elle peut conduire en effet au parcours du labyrinthe en entier avant de le résoudre. Il est possible de coder facilement un programme qui réalise cette technique. On dit que c'est un algorithme de parcours en profondeur de l'arbre. Nous ne nous intéresserons cependant pas ici à ce type d'algorithme.

La plupart du temps la résolution de labyrinthe repose sur la recherche du chemin le plus court afin d'arriver à la solution. On peut distinguer des situations qui amènent à deux techniques différentes de résolutions.

- La première situation est celle de la vue partielle du labyrinthe, c'est elle qui est intéressante si l'on veut résoudre un labyrinthe par un programme mais pas si l'on veut le résoudre à la main.

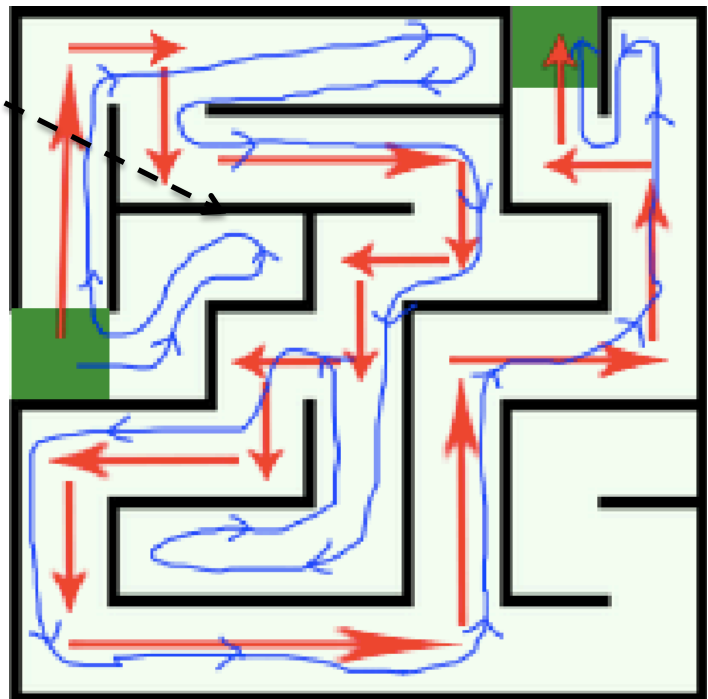
C'est en effet la situation dans laquelle serait une personne se trouvant à l'intérieur d'un labyrinthe sans vue d'ensemble. Cette personne ne disposerait alors d'aucune vue d'ensemble ni d'aucun repères.

Dans ce cas la technique du Fil d'Ariane est la plus efficace. Elle consiste à dérouler un fil sur son passage afin de pouvoir revenir en arrière lorsque l'on se trouve dans un cul de sac et permet ainsi d'explorer tous les chemins d'un carrefour jusqu'à trouver le bon.

*Le programme arrive dans un cul de sac et doit donc revenir en arrière*

**Légende :**

**Bleu :** Exemple de chemin parcouru par le programme  
**Rouge :** Chemin enregistré  
**Vers :** Entrée et sortie du labyrinthe.



C'est cette technique qui est utilisée dans la plupart des programmes permettant de trouver la solution des labyrinthes.



Ils consistent à incrémenter un pointeur qui va à l'aide d'une génération pseudo aléatoire de nombre correspondant aux murs, choisir des cases du labyrinthe à visiter. Il fera cela tant qu'il peut passer, donc tant qu'il n'y a pas de mur entre les cases, et ce jusqu'à ce qu'il prenne la valeur de l'arrivée.

Les cases déjà visitées seront enregistrées dans une liste, le programme leur donnera une valeur, par exemple 1.

Lorsque ces cases sont visitées une deuxième fois cela veut dire que le pointeur est revenu en arrière car il a rencontré un cul de sac, il va donc changer la valeur de ces cases, par exemple en 2.

A la fin il trace le chemin en mettant d'une autre couleur toutes les cases ayant ici la valeur de 1. (*voir schéma ci-dessus*).

On a alors le meilleur chemin pour résoudre un labyrinthe parfait maniériste.

- La deuxième situation est celle dans laquelle on a une vue d'ensemble du labyrinthe.

On est alors capable de distinguer géographiquement la sortie. On peut donc déduire des chemins en essayant de ce rapprocher de celle-ci et en revenant en arrière lorsque l'on tombe sur un cul de sac.

C'est la technique la plus utilisée pour la résolution à la main d'un labyrinthe.

## II/ La programmation d'un labyrinthe

### 1) La structure des labyrinthes en informatique

Comme on l'a vu précédemment un labyrinthe parfait maniériste peut être structuré en cellules. Un labyrinthe de  $m$  colonnes et  $n$  lignes comprend  $m \times n$  cellules qui seront reliées les unes aux autres par un chemin unique.

On peut voir dans le tableau ci-contre que les lignes et les colonnes sont numérotées, on donne donc à chaque case, des coordonnées. C'est comme cela que l'on représentera ces cellules en langage informatique.

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

Tableau représentant un labyrinthe 8x8.

Comme on peut voir ci-dessous, la topologie des surfaces connexes, c'est à dire des surfaces faites en un seul morceau, est identique quand celles-ci comportent le même nombre de cellules.

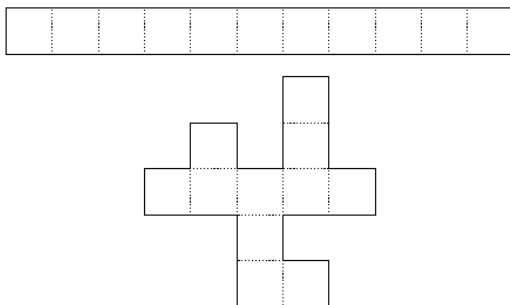
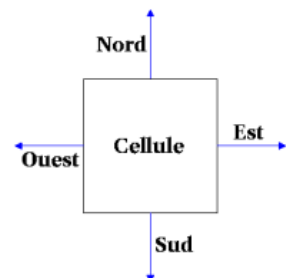


Schéma de surfaces de même topologie

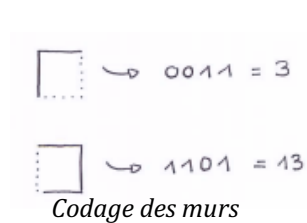
Dans le schéma ci-contre on peut voir 2 surfaces connexes différentes, cependant elles contiennent toutes les deux 11 cellules et 10 murs internes.

Chaque cellule est composée et limitée de 4 murs (ouverts ou fermés) selon les points cardinaux (Nord, Sud, Est, Ouest).

La plupart des algorithmes modélisent les cellules qui contiennent des murs et non les murs en eux-mêmes. Cependant nous verrons par la suite qu'il est possible de réaliser l'algorithme recherché en se basant sur la modélisation des murs.

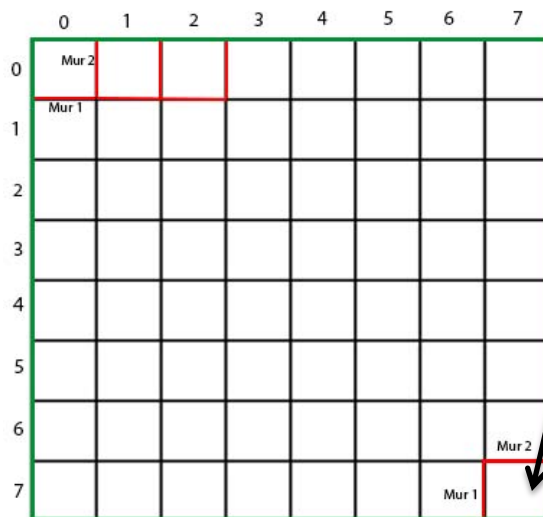


Cellule d'un labyrinthe



Pour chaque cellule, l'absence ou la présence de murs peut être codée sur 4 bits. En partant du mur Nord, puis Ouest, Sud et enfin Est.

On a un exemple sur le schéma ci contre où l'on peut voir sur quels bits sont codés chaque murs.



Pour obtenir le nombre total de murs dans un rectangle de  $m \times n$  cellules représentant le labyrinthe, on compte deux murs par cellule, moins le nombre de murs représentant la cellule en bas à droite.

Donc, le nombre de murs fermés dans un labyrinthe parfait est de :

$$2mn - m - n - (mn - 1) = (m - 1)(n - 1)$$

Pour le labyrinthe ci-dessus de 8 par 8, nous avons 49 murs internes.

Pour l'algorithme que nous avons créé nous avons décidé de réaliser un labyrinthe de 40 par 30 afin de répondre à la problématique de départ. En effet des analyses statistiques que nous expliciterons plus tard, nous ont révélé qu'un labyrinthe de 40 par 30 tracé par l'ordinateur prenait la plupart du temps plus d'1 minute à résoudre. Il y aura donc  $(40 - 1)(30 - 1) = 1131$  murs internes ouverts ou fermés.

La structure est maintenant posée. Voyons les différentes méthodes utilisant cette structure afin de programmer un générateur aléatoire de labyrinthes.

## **2) les différentes méthodes pour générer un labyrinthe**

La création d'un programme permettant de générer aléatoirement des labyrinthes peut s'avérer complexe. En effet la possibilité que le programme puisse donner un labyrinthe non conforme n'est pas acceptable. Il y a donc de très nombreux cas à traiter.

Ainsi la sélection et l'élaboration d'une méthode sont essentielles. C'est le premier point important de réflexion de notre projet de TIPE.

Voici ci-dessous toutes les méthodes sur lesquelles nous avons réfléchi avant d'aboutir à la méthode finale.

Elles sont classées de manière chronologique.

Les premières méthodes ont très vite été écartées nous nous sommes plutôt concentrés sur les 2 dernières. En effet, ce sont les seules qui répondent aux critères de notre problématique.

### 1.1) Première méthode étudiée

#### a) Principe

La première méthode, comme toutes les autres, se base sur la présence d'une grille et de cases qui ont chacune des coordonnées (voir la partie sur la structure des labyrinthes en informatique). Chaque case possède 4 murs.

Un mur entre deux cases est donc composé de deux murs .

L'algorithme passe de case en case depuis le début jusqu'à la fin de la ligne (ou de la colonne) puis passe à la ligne suivante (ou de la colonne).

À chaque fois qu'il passe dans une case, il décide aléatoirement d'ouvrir une ou plusieurs portes.

L'algorithme s'arrête lorsqu'il a parcouru toutes les cases du tableau.

### b) Points positifs

Le programme est très simple à mettre en place.

Il ne demande pas des connaissances très pointues en informatique. De plus la manière de mettre en place les différentes fonctions du programme, est aussi très facile.

### c) Points négatifs

Le labyrinthe créé par ce programme a de grandes chances de ne pas être un labyrinthe parfait.

De plus, on peut se retrouver avec une feuille blanche ou une simple grille si le programme choisit de supprimer tous les murs, ou au contraire de les garder tous.

De même le chemin de résolutions peut être multiple, comme ne pas exister du tout.

Enfin le fait de passer d'une case à l'autre de manière prédéfinie pose certains problèmes.

En effet le mur entre chaque case est double (il y a le mur assigné à la case de départ et le mur associé à sa voisine. Visuellement il n'y a qu'un seul mur, alors qu'en fait, ce sont deux murs l'un sur l'autre).

Le problème est qu'il est difficile de dire au programme de supprimer le mur de la case à côté s'il n'a pas accès à ses coordonnées !

En effet rien n'indiquera au programme si les murs de la case précédente ont été supprimés ou non.

Nous avons donc écarté cette méthode.

## 1.2) Deuxième méthode étudiée

### a) Principe

On améliore la méthode précédente en forçant le programme à avoir au moins un chemin de résolution.

Pour ce faire, on va ajouter une valeur booléenne aux cases du tableau qui dira si la case a été oui ou non déjà changé par le programme.

Le programme part de la case en haut à gauche, puis choisit une direction : en bas ou à droite. Il supprime ensuite les murs pour passer à la case suivante (mur de la case de départ et de la case d'arrivée pour contrer le problème des doubles murs).

Arrivé sur la seconde case il choisit à nouveau à droite ou en bas, afin d'aller sur une autre case et ainsi de suite jusqu'à ce qu'il arrive à la case en bas à droite.

Si on se retrouve sur le bord de la grille sans être à la sortie alors :

-Si il est sur la dernière colonne, le programme n'ouvrira plus que les portes du bas

-Si il est sur la dernière ligne, le programme n'ouvrira plus que les portes de droite

Une fois que ce chemin est effectué, le programme va visiter toutes les cases du tableau qui n'ont pas encore été parcourues, soit ligne par ligne, soit colonne par colonne. Dès qu'on arrive dans une case le programme supprime ou non les murs. Le programme s'arrête dès que toutes les cases du tableau ont été visitées. Donc comme pour le programme précédent, on part d'une grille. Dans chaque case il y a les coordonnées des quatre murs qui l'entoure et en plus une valeur booléenne. Celle-ci dit si la case a été visitée ou non. Au début toutes les valeurs booléennes sont sur « faux » (la case n'a pas été visitée), puis le programme s'arrête quand elles sont toutes sur « vrai ».

### b) Points positifs

Le programme est plutôt simple à mettre en place. Les connaissances requises pour créer les fonctions du programme sont basiques. De plus le labyrinthe possède au moins un chemin et est donc possible à résoudre.

### c) points négatifs

On retrouve les mêmes problèmes qu'avec le labyrinthe précédent, à l'exception que le labyrinthe possède un chemin et sera donc résolvable. Mais de même qu'avec le précédent algorithme, le labyrinthe a de grandes chances d'être imparfait. De plus le chemin aura toujours plus ou moins la même forme, c'est à dire qu'il faudra aller à droite ou en bas pour se retrouver à la sortie. Aussi, on ne peut pas se retrouver avec une grille ou une page blanche, mais par contre on peut très bien se retrouver avec un chemin au milieu d'une page blanche ou d'une grille, ce qui ne change pas grand-chose. Le labyrinthe n'en est donc plus un si on se retrouve avec ce genre de génération. De plus on retrouve exactement les mêmes problèmes de murs doubles que dans le programme précédent à l'exception des cases du chemin qui sont elles, générées de manière forcée.

### 1.3) Troisième méthode envisagée

#### a) principe

Cette méthode est faite pour se débarrasser des murs doubles et donc de tous les problèmes et de la complexité que la présence de ces derniers implique.

Nous avons choisi de mettre en pratique une méthode avec laquelle il n'y a qu'un mur entre chaque case.

On a une grille de départ.

Tous les murs extérieurs ne sont que des dessins. Le programme n'a aucun pouvoir sur eux.

Chaque case possède un mur du bas et un mur de droite ainsi qu'une valeur (booléenne) indiquant si elle a été visitée ou non.

Le programme s'exécute de cette manière :

Il part de la case de départ que l'on fixe en haut à gauche dans notre cas.

Il passe de case en case jusqu'à ce qu'il tombe sur un côté du labyrinthe. La case sera alors immédiatement considérée comme la case de sortie.

Le labyrinthe passera ensuite de case en case non visitée, en case non visitée et supprimera ou non les murs toujours de manière aléatoire.

Le programme s'arrête quand toutes les cases ont été visitées.

#### b) Points positifs

Nous avons enfin réussi à nous débarrasser des problèmes de murs doubles.

De plus la mise en place du programme reste simple.

#### c) Points négatifs

Nous nous sommes débarrassés des problèmes de murs doubles, mais le labyrinthe n'est, dans la plupart des cas, toujours pas parfait. De plus le chemin qui est généré est forcément trop simple.

En effet, celui-ci ne peut très bien faire que seulement quatre cases de long et dans tous les cas il ne parcourra que très rarement plus de la moitié de la largeur du labyrinthe.

Enfin on s'est débarrassé des murs doubles, mais cela pose un problème.

Si le programme est dans une case et veut aller en haut ou à gauche, comment va t'il faire pour supprimer le mur ?

En effet les coordonnées de celui ci sont dans la case de gauche ou du haut et non dans la case dont s'occupe le programme à ce moment.

Il faut donc créer un nouveau système qui relie les cases entre elles indépendamment des murs.

Ce qui rajoute tout un tas de coordonnées qu'on rassemble dans une liste à gérer en parallèle de la liste des cases.

#### 1.4) Quatrième méthode envisagée

##### a) principe

On se débarrasse des cases et on ne garde que des murs.  
On a donc deux types de murs :

- les horizontaux
- les verticaux.

Pour former la grille les murs sont donc répartis en colonnes pour les murs horizontaux et en lignes pour les murs verticaux.

On crée une structure pour chaque murs.

En plus du type horizontal ou vertical, chaque structure aura deux coordonnées et une valeur booléenne qui indiquera si le programme a déjà effectué une opération sur le mur.

Le programme choisit aléatoirement un mur dans la grille, le supprime ou non, puis passe à un autre mur du même type.

Dés qu'il s'est occupé de tous les murs du même type (quand les valeurs booléennes sont sur "vrai" pour tous les murs verticaux ou horizontaux), le programme passe aux murs de l'autre type.

Le programme s'arrête lorsqu'il a effectué l'opération aléatoire de suppression sur tous les murs.

##### b) Points positifs

L'algorithme est toujours simple à mettre en place. De plus on se débarrasse des murs doubles sans se créer d'autres difficultés.

Le fait de ne pas utiliser de cases rend cette méthode facile à programmer.

En effet, il n'est pas nécessaire d'avoir des cases qui contiennent des coordonnées ce qui facilite le travail informatique.

##### c) Points négatifs

Le labyrinthe créé par ce programme a de grandes chances de ne pas être un labyrinthe parfait.

De plus on peut même se retrouver avec une feuille blanche ou une grille si le programme choisit de supprimer tous les murs ou au contraire de les garder tous.



### 1.5) Cinquième méthode étudiée

#### a) Principe:

Cette méthode utilise une propriété des labyrinthe parfaits qui est que toutes les cases du labyrinthe sont reliées aux autres.

Le principe est le suivant: on part d'une grille.

Chaque case du labyrinthe a une valeur qui est différente de celle de toutes les autres cases.

Toutes les portes sont fermées au départ.

Le programme choisit une case au hasard. Puis il cherche une case à coté d'elle qui n'a pas la même valeur puis il ouvre le mur entre ces deux cases.

Dès qu'une porte est ouverte entre deux cellules adjacentes, la valeur de la première case devient aussi celle de la deuxième.

Puis le programme choisit aléatoirement une autre case et recommence.

On n'ouvre des portes seulement entre deux cellules qui n'ont pas la même valeur.

Le choix des portes s'effectue de manière aléatoire.

L'algorithme se présentera donc sous la forme d'une boucle qui tente d'ouvrir aléatoirement des portes.

#### b) Points positifs

Cette méthode est très complète et permet la génération d'un labyrinthe parfait.

De plus elle est facilement compréhensible sa difficulté ne demeurant pas dans la mise en oeuvre de l'algorithme mais dans la maîtrise d'éléments informatique complexe.

#### c) Points négatifs

Cet algorithme est complet mais la méthode utilisée demande une très grande connaissance en informatique du fait de l'utilisation de plusieurs listes doublements chaînées.

De plus le choix des murs étant aléatoire, il faut gérer une liste des murs disponibles tout le temps pour ne pas tomber sur un mur déjà ouvert ou un mur qu'on ne peut pas ouvrir (entre deux cases de même valeur).

Sinon on risque de prendre un temps infini pour générer le labyrinthe.

On peut essayer de régler ce problème en faisant une analyse statistiques du nombre de fois que la boucle doit être parcourue pour générer le labyrinthe. Mais on ne sera jamais sûr à 100% que le labyrinthe sera fonctionnel.

## Schéma de la méthode n°5

Le programme choisit au hasard un mur à casser.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Les deux cases adjacentes à ce mur prennent la même valeur, puis le programme choisit un autre mur.

0	1	2	3	4
5	5	7	8	9
10	11	12	13	14
15	16	17	18	19

Le programme répète l'opération.

0	1	2	3	4
5	5	7	8	9
10	11	12	13	14
15	16	17	18	19

Le programme continue de casser des murs et de donner aux cellules adjacentes à ces murs la même valeur.

0	1	2	2	4
5	5	7	2	9
10	11	12	2	14
15	11	17	17	19

Le programme arrive sur un mur donnant sur une case déjà visitée, mais n'ayant pas la même valeur que la case sur laquelle il est, il casse donc tout de même ce mur.

0	1	2	2	4
5	5	5	2	9
10	11	12	2	14
15	11	17	17	19

Le programme donne aux deux ensembles de cases la valeur du premier ensemble

0	1	5	5	4
5	5	5	5	9
10	11	12	5	14
15	11	17	17	19

Le programme continue sur sa lancée et de plus en plus de cases ont la même valeur.

0	0	5	5	4
5	5	5	5	5
5	5	12	5	14
15	5	12	12	14

Finalement toutes les cases ont la même valeur. Le labyrinthe est prêt.

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

### 3) La méthode choisie : Explication du principe et du programme






#### a. Explication rapide du principe

Le schéma ci-dessous résume parfaitement la construction du labyrinthe par l'algorithme que nous avons choisi.

On part d'un quadrillage. Le programme choisit (en partant du début en haut à gauche) un mur au hasard. S'il est cassable (c'est le cas quand il n'est pas un bord du labyrinthe ou qu'aucune case déjà visitée se trouve derrière celui-ci) il le casse, sinon il en choisit un autre jusqu'à en trouver un qui soit cassable. S'il ne trouve aucun mur cassable, il revient en arrière jusqu'à trouver un mur cassable. Il continue ensuite jusqu'à revenir sur la case de départ. (Cf schéma)

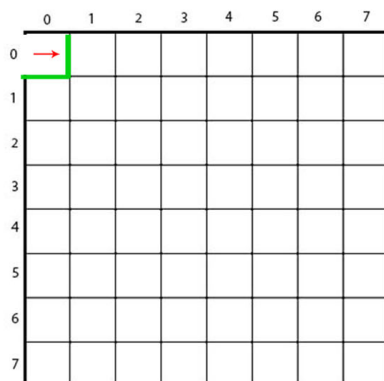
### Schéma de la construction du labyrinthe par le programme :

#### Légende :

-  Murs indestructibles car case visité des deux côtés.
-  Chemins choisis au hasard par le programme.
-  Frontières du labyrinthe, murs indestructibles.
-  Murs destructibles.
-  Murs que le programme peut choisir au hasard.

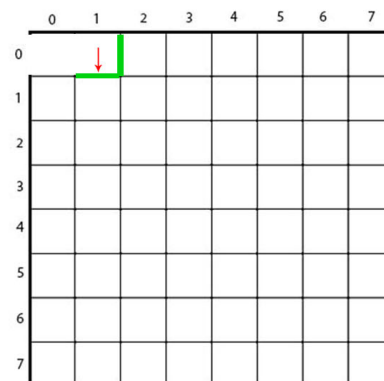
#### Etape 1 :

Le programme part de la case au début du labyrinthe, et choisit au hasard un des 4 murs de cette case. S'il est incassable ou bien si c'est le départ il retire au hasard un des 3 murs restant. Il continue jusqu'à tirer un mur cassable ( sur le schéma un mur en **vert** )



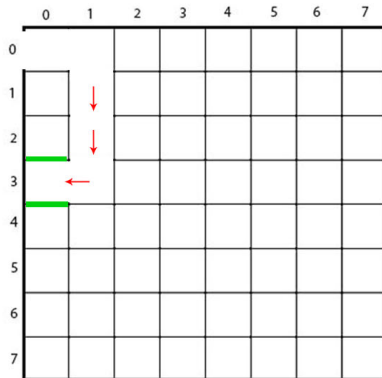
#### Etape 2 :

Le programme détruit le mur choisit au hasard puis répète l'opération de l'étape 1 jusqu'à tomber sur un mur cassable.



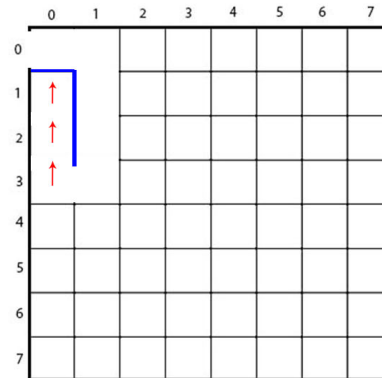
### Etape 3 :

Le programme continue à choisir des murs au hasard et à les casser comme il l'a fait dans l'étape 1 et 2



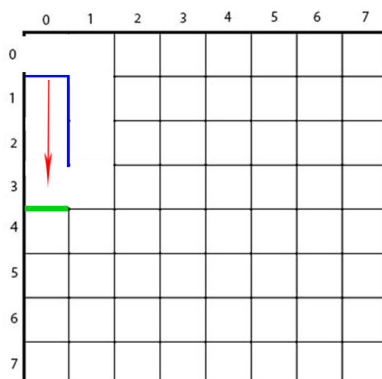
### Etape 4 :

Le programme continue à choisir des murs au hasard et à les casser jusqu'à arriver à une case entourée de murs incassables. Ils peuvent être sincassables car donnent sur une case déjà visitée (en Bleu) ou car ce sont des bords du labyrinthe. Il va alors faire demi-tour.



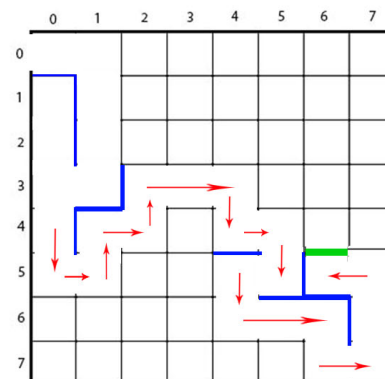
### Etape 5 :

Après avoir fait demi-tour, le programme revient sur ses pas jusqu'à trouver un mur cassable.



### Etape 6 :

Le programme continue alors son chemin même s'il arrive à la fin du labyrinthe. Il continue à casser des murs jusqu'à revenir sur la case de départ.



## b. Explication du programme.

**NB :** Durant l'explication, il apparaîtra par moment des fonctions telles que `allocationMemoire(x, y)` afin de créer des tableaux. Elles ne seront pas détaillées dans ce dossier puisqu'elles n'aident nullement à la compréhension de l'algorithme. Elles seront toutefois affichées dans l'annexe car nécessaires à l'exécution du code. Ce qu'il faut retenir est que ces fonctions réservent de la place dans la mémoire de l'ordinateur pour créer les différents tableaux nécessaires à la création du labyrinthe, dont le tableau du labyrinthe.

On crée un objet case. Cet objet contient les valeurs des 4 côtés représentant la case, plus un booléen qui est "vrai" si la case est déjà visitée.

Lors de l'initialisation, tous les côtés de la case valent 1. Si un côté vaut 1, alors ce côté contient un mur. Si un côté vaut 2, alors ce côté contient un mur "indestructible" (Ce sont les bords du labyrinthe). Sinon, si un côté vaut 0, alors ce côté sera un chemin où l'on pourra se déplacer.

```
struct Case{  
  
    // Les côtés de la case  
    int haut;  
    int bas;  
    int gauche;  
    int droite;  
  
    // Booléen si la case a été visitée  
    int caseVisite;  
  
};
```

On va ensuite créer une case « type » qui servira de base à toutes celles du labyrinthe. Pour l'instant, tous ses côtés vaudront 1, c'est-à-dire que tous les côtés posséderont un mur. Nous modifierons ensuite chaque case séparément afin de créer le chemin du labyrinthe.

```
// x et y sont l'abscisse et l'ordonnée maximale d'une case du labyrinthe  
int x, y; x = 30; y = 30;  
// On alloue la mémoire nécessaire pour la création du labyrinthe  
// Ce labyrinthe aura x cases en abscisse et y case en ordonnées  
struct Case **labyrinthe = allocationMemoire(x, y);  
  
// On crée un case "type"  
    struct Case maCase;  
    maCase.haut = 1;  
    maCase.bas = 1;  
    maCase.gauche = 1;  
    maCase.droite = 1;  
    maCase.caseVisite = 0;
```

A présent, nous allons attribuer la valeur de la case « type » à chaque case de notre labyrinthe. On prendra également soin de modifier la valeur des murs définissant les

bords du labyrinthe en leur attribuant la valeur 2. Ainsi, nous les rendrons « indestructibles » lors de la construction de notre labyrinthe.

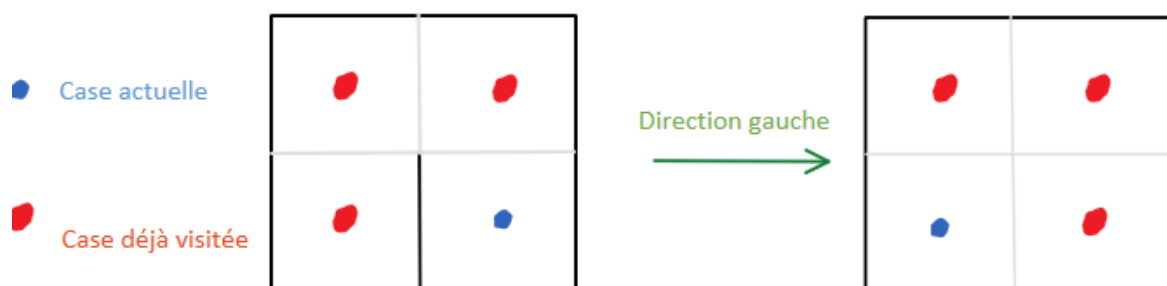
```
// On affecte à chaque case du labyrinthe la valeur de la case type
for(i = 0; i < x; i++){
    for(j = 0; j < y; j++){
        labyrinthe[i][j] = maCase;
    }
}

/*
On change la valeurs des murs définissant les bords du labyrinthe
afin d'en faire des murs indestructibles.
*/
for(i = 0; i < x; i++){ // Les bords du haut et du bas
    labyrinthe[i][0].haut = 2;
    labyrinthe[i][y-1].bas = 2;
}

for(j = 0; j < y; j++){ // Les bords de droite et de gauche
    labyrinthe[0][j].gauche = 2;
    labyrinthe[x-1][j].droite = 2;
}
```

A partir d'ici, le labyrinthe est correctement initialisé et il est de  $x*y$  cases. Les bords sont "indestructibles", on ne pourra pas les modifier par erreur leur de la construction. Pour le moment, chaque case est dotée de 4 murs. On a donc en fait un quadrillage. Nous commençons véritablement la construction du labyrinthe maintenant.

Nous choisirons d'abord une case au hasard comme point de départ. Ensuite, nous essayerons de « casser » le mur dans une direction donnée : s'il ne s'agit pas d'un des bords du labyrinthe et que nous ne sommes pas déjà passés sur la case adjacente, alors on casse le mur dans la direction choisie. En effet, si nous ne vérifions pas si nous avons déjà visité la case adjacente, nous aurions pu casser un des murs définissant un chemin :



Nous finirons quel que soit la dimension avec un carré, « détruisant » ainsi notre labyrinthe.

Si sur une case donnée, aucune des directions n'est possible, alors on revient sur la case précédente pour essayer de casser de nouveau un des murs sur cette case-là. Si cela n'est toujours pas possible, nous revenons encore une fois en arrière sur la case d'encore avant et ainsi de suite jusqu'à trouver un mur cassable.

S'il n'y en a plus, nous reviendrons à un moment donné à la case de départ : nous considérerons alors que la construction du labyrinthe est finie.

Nous allons d'abord créer deux tableaux, un qui contient l'historique des abscisses, l'autre celui des ordonnées : dans le cas où aucune direction n'est possible, alors on revient sur nos pas à l'aide de ces deux tableaux. Nous créons aussi deux variables qui contiendront la longueur des tableaux associés. Nous aurions pu n'en créer qu'une pour les deux tableaux, mais nous faisons ainsi pour faciliter la compréhension.

```
int *historiqueX;
int *historiqueY;

/* Pour l'instant, ces tableaux ne contiennent aucune position, inutile
d'allouer de la mémoire, mais il nous faut l'initialiser */

historiqueX = tableauHistorique(1, 0);
historiqueY = tableauHistorique(1, 0);
// Les longueurs des tableaux d'historique. Pour le moment elles sont à 0
int longueurX = 0, longueurY = 0;
```

tableauHistorique est une fonction qui va créer le tableau contenant l'historique des abscisses ou des ordonnées. On préfère faire un tableau pour les abscisses et un pour les ordonnées plutôt qu'un seul à deux dimensions afin de faciliter le parcours des positions. En effet, il n'existe pas en C des objets définissant un couple comme en Python avec les tuples.

Cette fonction reçoit en paramètre la taille du tableau ainsi que la valeur de la position actuelle.

On choisit au hasard les coordonnées *x* et *y* de la case où nous allons commencer à construire le labyrinthe.

```
srand(time(NULL)); // Fonction de génération pseudo aléatoire
int xi = rand() % x;
int yi = rand() % y;
```

Nous commençons ensuite à construire le labyrinthe, nous entrons dans une boucle. Chaque tour de boucle correspond à une nouvelle position dans le labyrinthe. `construire` est une variable qui vaut 1 tant que l'on n'a pas fini de construire notre labyrinthe.

Nous créons un tableau qui contient nos directions. Il se réinitialise à chaque tour de boucle, c'est-à-dire à chaque fois qu'on est sur une nouvelle case.

```
while(construire == 1){

    int *direction;
    direction = tableauHistorique(4, 0);

    int i;
    for(i = 0; i < 4; i++){
        direction[i] = i;
    }
    int directionsPossibles = 4;

    [...]
```

A l'intérieur de cette boucle, nous en créons une deuxième. Chaque tour de cette boucle correspondra à un essai dans une direction. `onATrouveUneDirection` est une variable booléenne qui est "vraie" (= 1) lorsqu'on a trouvé une direction dans laquelle on peut aller.

```

while(onATrouveUneDirection == 0){
    // Nous sommes sur un case, on l'a alors visitée.
    labyrinthe[xi][yi].caseVisite = 1;

    /* On crée deux nouveaux tableaux qui contiennent l'abscisse et l'ordonnée
    actuelle */
    int *actuelX, *actuelY;

    actuelX = tableauHistorique(1, xi);
    actuelY = tableauHistorique(1, yi);

```

Nous avons créé deux nouveaux tableaux pour enregistrer notre position actuelle dans le cas où il faudrait revenir sur cette position sur la prochaine case si aucune direction n'est possible. Pour cela on concatène le tableau des historiques avec le tableau des abscisses et des ordonnées de la nouvelle position.

Nous créons de nouveaux tableaux car il n'est pas possible en C d'ajouter simplement une variable dans un tableau. Il faut l'ajouter dans un nouveau tableau que nous additionnerons avec l'historique existant. En effet, nous pouvons sommer deux tableaux, mais nous ne pouvons pas sommer un tableau et une variable.

```

historiqueX = additionTableau(historiqueX, actuelX, longueurX);
historiqueY = additionTableau(historiqueY, actuelY, longueurY);
longueurX += 1; // On a rajouté la position actuelle dans le tableau des
historiques la longueur a donc augmentée
longueurY += 1; // Pareil

```

On choisit ensuite au hasard la direction dans laquelle on va commencer à aller casser du mur. Quel que soit la direction où l'on va, on vérifie que le mur n'est pas un des bords du labyrinthe et que la case n'a pas déjà été visitée.

Seul le cas où la direction choisie est le haut est copié ici. Dans le code final, il existe aussi un **case** BAS, **case** DROITE et **case** GAUCHE qui sont exactement les mêmes, excepté les modifications concernant les `xi` et `yi` selon la direction.

```

int onVaDansCetteDirection = direction[rand() % directionsPossibles];

switch(onVaDansCetteDirection){

    case HAUT:
if(labyrinthe[xi][yi].haut == 2 || labyrinthe[xi][yi-1].caseVisite == 1){

    // S'il y a un mur incassable dans cette direction ou que la case a déjà
    été visitée, on crée une copie de notre tableau de direction actuel moins
    la direction testée

    direction = copieDirections(direction, directionsPossibles, HAUT);

    // On a retire la direction actuelle qui ne fonctionne pas
    directionsPossibles -= 1;

    longueurX -= 1; // On ne bouge pas, on ne rajoute pas la position à
    l'historique...
    longueurY -= 1; // ... donc sa taille ne varie pas.

```



```

break;
// Le break est un mot clé qui signifie qu'on a fini de tester notre case
}

else{ // Sinon, on peut aller dans cette direction !
labyrinthe[xi][yi].haut = 0; // On détruit le mur sur la case actuelle...
labyrinthe[xi][yi - 1].bas = 0; // ... qui est le même que sur la case
adjacente.
yi -= 1; // On se place sur la nouvelle case
onATrouveUneDirection = 1; // Hé oui ! Il faut sortir de la boucle si on a
une direction
break;
}

```

Nous décrémentation longueurX et longueurY car celles-ci ont été incrémentées au début de la boucle qui teste la direction choisie.

Toujours dans la boucle, après avoir testé la direction, nous regardons si nous n'avons pas testé les quatre possibles. Si aucune n'est possible, alors il faut revenir en arrière. On regarde d'abord si cette position antérieure n'est pas celle de départ. Si c'est le cas, nous avons fini de construire le labyrinthe, donc on sort de la boucle de construction.

```

if(directionsPossibles == 0){
    xi = historiqueX[longueurX - 1]; // xi prends la valeur du dernier
    élément dans l'historique
    yi = historiqueY[longueurY - 1]; // Pareil

    if(xi == historiqueX[0] && yi == historiqueY[0]){
        construire = 0; // On arrête la construction, on a fini
    }
}

```

Il faut supprimer la position qui n'a pas marché de l'historique pour ne pas revenir dessus si la position précédente ne marche pas : Soit N la dernière position et N-1 l'avant dernière. Si N ne marche pas on revient sur N-1. Si on ne supprime pas N, N devient N-1 et N-1 devient N.

On crée alors une boucle infinie.

```

historiqueX = soustractionHistorique(historiqueX, longueurX);
historiqueY = soustractionHistorique(historiqueY, longueurY);
longueurX -= 1;
longueurY -= 1;

/* Même si c'est pas tellement vrai mais il faut sortir de la boucle
On sait où on va : en arrière */
onATrouveUneDirection = 1;

```

Une fois que la première boucle est finie, notre labyrinthe est alors créé. Il nous reste à l'afficher. Nous déciderons que les cases sont de dimensions 20\*20 pixels.

Pour cela, on crée une ligne type et une colonne type à l'aide de la bibliothèque SDL, cette bibliothèque étant très répandue en C et C++ pour des représentations graphiques en 2D.

```

SDL_Surface *lignes;
SDL_Surface *colonnes;

lignes = SDL_CreateRGBSurface(SDL_HWSURFACE, 20, 2, 32, 0, 0, 0, 0);
SDL_FillRect(lignes, NULL, SDL_MapRGB(ecran->format, 0, 0, 0));

colonnes = SDL_CreateRGBSurface(SDL_HWSURFACE, 2, 20, 32, 0, 0, 0, 0);
SDL_FillRect(colonnes, NULL, SDL_MapRGB(ecran->format, 0, 0, 0));

```

Les paramètres envoyés à `SDL_CreateRGBSurface` sont, dans l'ordre :

- En quelle mémoire (Vidéo ou Système) sera chargée la surface (notre trait)
- La largeur de la surface
- La longueur de la surface
- Le nombre de couleur
- Le code couleur pour le rouge, le vert et le bleu
- Le degré de transparence de notre surface

Les paramètres envoyés à `SDL_FillRect` sont :

- La surface à colorer
- La partie de la surface à colorer. En envoyant `NULL`, nous signifions que nous voulons colorer l'intégralité de notre trait
- La couleur. `SDL_MapRGB` renvoie une couleur, ici (0,0,0) étant du noir.

Egalement, on crée une variable qui contiendra les coordonnées de l'origine de nos lignes/colonnes.

```

SDL_Rect position;
position.x = 50;
position.y = 50;

```

Nous allons travailler sur chaque case séparément. Si le mur X de cette case ne vaut pas 0, c'est-à-dire n'est pas un chemin, alors c'est un mur et il faut donc tracer la colonne ou la ligne représentant ce mur.

Pour parcourir l'ensemble des cases, nous utiliserons deux boucles `for`, l'une qui parcourra les colonnes, l'autre les lignes.

Les fonction `SDL_BlitSurface` « tracent » les lignes/colonnes sur la fenêtre. Elles prennent en paramètres :

- La surface à tracer.
- La partie de la surface à tracer. En envoyant `NULL`, nous signifions que nous voulons tracer l'intégralité de notre trait.
- La surface dans laquelle on doit tracer, ici il s'agit de notre écran.
- Les coordonnées de l'origine de la surface

```

for(j = 0; j < y; j++){
    position.x = 50;
    // Sur chaque lignes :
    for(i = 0; i < x; i++){ // On parcourt les colonnes
        // On travaille donc à partir d'ici sur la case (i,j)

        /*
            On regarde maintenant les 4 murs de la case. Si l'un
            d'entre eux vaut 0, on ne tracera pas de trait
        */
        if(labyrinthe[i][j].haut != 0){
            SDL_Blitsurface(lignes, NULL, ecran, &position);
        }
        if(labyrinthe[i][j].gauche != 0){
            SDL_Blitsurface(colonnes, NULL, ecran, &position);
        }
        if(labyrinthe[i][j].bas != 0){
            SDL_Rect positionBas;
            positionBas.x = position.x;
            positionBas.y = position.y + 20;

            SDL_Blitsurface(lignes, NULL, ecran, &positionBas);
        }
        if(labyrinthe[i][j].droite != 0){
            SDL_Rect positionDroite;
            positionDroite.x = position.x + 20;
            positionDroite.y = position.y;

            SDL_Blitsurface(colonnes, NULL, ecran, &positionDroite);
        }
        position.x += 20;
    }
    position.y += 20;
}

```

Après avoir tracé les murs d'une case dans la deuxième boucle for, nous incrémentons `position.x` de 20. En effet, au prochain tour de cette même boucle, nous allons travailler sur la case « à droite », dont l'origine est décalée de 20 pixels par rapport à la première case.

Même raisonnement pour `position.y`, dès que nous avons tracé toutes les cases de la  $j^{\text{ième}}$  ligne, nous « descendons », on incrémente l'ordonnée de 20.

En programmation, l'origine (0,0) se situe en haut à gauche de l'écran et les ordonnées se comptent « en descendant ».

Petite particularité pour les murs de droite et du bas : nous passons par une autre position que celle de départ. (`positionDroite` ou `positionBas`). En effet, prenons l'exemple du mur de droite.

Sa position est décalée de 20 pixels par rapport au mur de gauche.

Nous ne pouvons pas envoyer séparément l'abscisse et l'ordonnée à l'origine à la fonction `SDL_Blitsurface`. Nous ne pouvons qu'envoyer les coordonnées (l'abscisse et l'ordonnée) à notre fonction.

C'est pourquoi nous créons une nouvelle coordonnée à l'origine décalée dans le sens voulu. C'est cette coordonnée que nous enverrons à la fonction.

A la sortie de la deuxième boucle for, le labyrinthe est entièrement tracé !

c. Analyse statistique du temps nécessaire pour résoudre les labyrinthe générés par le programme.

Afin de répondre à notre problématique de départ. Nous devons vérifier si dans la plupart des cas les labyrinthes produits par notre programme mettent plus d'une minute à être résolus. Pour cela nous avons fait une analyse statistique sur un échantillon de 10 personnes réalisant 3 labyrinthes différents. ( les 30 labyrinthes testés sont tous différents).  
Voici les résultats de cette analyse.

Personnes essayant de résoudre la labyrinthe		Temps que ces personnes ont mises.
Personne 1	Test 1	2.50min
Personne 1	Test 2	0.46min
Personne 1	Test 3	0.57min
Personne 2	Test 1	1.32min
Personne 2	Test 2	1.15min
Personne 2	Test 3	1.16min
Personne 3	Test 1	0.58min
Personne 3	Test 2	0.49min
Personne 3	Test 3	1.56min
Personne 4	Test 1	2.23min
Personne 4	Test 2	2.27min
Personne 4	Test 3	1.18min
Personne 5	Test 1	2.12min
Personne 5	Test 2	1.05min
Personne 5	Test 3	2.40min
Personne 6	Test 1	1.47min
Personne 6	Test 2	3.12min
Personne 6	Test 3	2.56min
Personne 7	Test 1	1.01min
Personne 7	Test 2	0.46min
Personne 7	Test 3	0.58min
Personne 8	Test 1	1.34min
Personne 8	Test 2	1.05min
Personne 8	Test 3	1.56min
Personne 9	Test 1	0.57min
Personne 9	Test 2	2.11min
Personne 9	Test 3	1.31min
Personne 10	Test 1	2.54min
Personne 10	Test 2	1.32min
Personne 10	Test 3	1.03min

La valeur minimal est :  $\text{Min} = 0.46 \text{ min}$

La valeur Maximale est :  $\text{Max} = 3.12 \text{ min}$

Soit S la somme de tout les temps :

$$S = 42.04$$

Soit n le nombre de test :  $n = 30$

La moyenne du temps mis par ces 30 personnes pour résoudre le labyrinthe est donc de :  $M = S/n$

$$M = 42.04/30 = 1.40 \text{ min}$$

Ainsi les individus mettent en moyenne 1.40min pour résoudre le labyrinthe ce qui convient pour répondre à notre problématique.

## **Bibliographie :**

- [http://www.di.ens.fr/~pointche/stages/projets/ensta04\\_1.pdf](http://www.di.ens.fr/~pointche/stages/projets/ensta04_1.pdf)
- <http://www.enseignement.polytechnique.fr/informatique/profs/Laurent.Viennot/info1B/2003-2004/TD4/index.html>
- <http://ilay.org/yann/articles/maze/>
- <http://codes-sources.commentcamarche.net/forum/affich-1284500-algorithme-generer-un-labyrinthe-aleatoire>
- <http://www.encyclopedie-incomplete.com/?Modelisation-et-Creation-d-un>
- [https://fr.wikipedia.org/wiki/Mod%C3%A9lisation\\_math%C3%A9matique\\_d%27un\\_labyrinthe](https://fr.wikipedia.org/wiki/Mod%C3%A9lisation_math%C3%A9matique_d%27un_labyrinthe)
- <https://fr.wikipedia.org/wiki/Labyrinthe>
- Nous avons appris à coder C à l'aide de ce tuto sur openclassroom :  
<https://openclassrooms.com/courses/apprenez-a-programmer-en-c/vous-avez-dit-programmer>