Projet Labyrinthe
Création et exploration de labyrinthes

Résumé

Le but du projet est d'implémenter la génération de différents labyrinthes de diverses formes et de les parcourir avec des algorithmes de recherches de chemin.

Groupe de projet

- Geoffrey HAON
- Anne FELTEN

Structure de données

Dans le cadre de ce projet, l'implémentation de différentes structures a été nécessaire

- Des **matrices** : pour contenir la carte des différents labyrinthes.
- Des **ensembles** : pour contenir des ensembles de données utiles dans les algorithmes.
- Des **Labyrinthes** : pour contenir les informations sur les labyrinthes.
- Un **Tas Binaire** (Heap) : pour l'optimisation de d'algorithme de tri.

• Matrices

La matrice simple est un tableau d'entiers stockés sur une dimension mais dans laquelle on peut faire des traitements d'un tableau de deux dimensions et ainsi gérer une carte de deux dimensions.

On a aussi créé deux variantes de la matrice :

- **MatriceCarre** : pour le labyrinthe avec des murs carrés et pour le labyrinthe circulaire
- **MatriceHexa** : pour le labyrinthe hexagonal

```
typedef struct Matrice{

  int h;
  int l;
  int * points;

} Matrice;
```

```
typedef struct MurCarre{

        int c1; // coté |
        int c2; // coté _

        int v;

} MurCarre;

typedef struct MatriceCarre{

        int h;
        int l;
        MurCarre* * points; /

} MatriceCarre;
```

```
typedef struct MurHexa{

        int c1; // coté |
        int c2; // coté /
        int c3; // coté \

        int v;

} MurHexa;

typedef struct MatriceHexa{

        int h;
        int l;
        MurHexa* * points;

} MatriceHexa;
```

- ## Ensemble

L'ensemble est une liste doublement chainée avec une sauvegarde du premier nœud et du dernier nœud.

De cette façon il est très facile de parcourir la liste pour accéder à l'index d'un nœud ou faire une recherche. La complexité de parcours est au plus O(n/2) si on connait la position de l'élément à trouver.

Deux variantes ont été faites pour les labyrinthes à murs carrés, hexagonal et circulaire.

```c
/* Noeud de l'ensemble */
typedef struct Noeud{
    /* data*/
    int x; // [x,y]
    int y;

    /* list suite */
    struct Noeud * next ;
    struct Noeud * previous;
}Noeud;

/* Ensemble */

typedef struct Ens{
    int taille;
    struct Noeud * premier;
    struct Noeud * dernier;
}Ens;
```

```c
/* Noeud de l'ensemble */
typedef struct NoeudCarre{
    /* data*/
    int x; // [x,y]
    int y;
        int cote;

    /* list suite */
    struct NoeudCarre * next ;
    struct NoeudCarre * previous;
}NoeudCarre;

/* Ensemble */

typedef struct EnsCarre{
    int taille;
    struct NoeudCarre * premier;
    struct NoeudCarre * dernier;
}EnsCarre;
```

```c
/* Noeud de l'ensemble */
typedef struct NoeudHexa{
    /* data*/
    int x; // [x,y]
    int y;
    int cote;//

    /* list suite */
    struct NoeudHexa * next ;
    struct NoeudHexa * previous;
}NoeudHexa;

/* Ensemble */

typedef struct EnsHexa{
    int taille;
    struct NoeudHexa * premier;
    struct NoeudHexa * dernier;
}EnsHexa;
```

Remarque : En effet, **l'Ensemble Carre** est identique à **l'Ensemble Hexagonal**. Cependant, ces deux ensembles ont été développés en parallèle et conservés afin de maintenir un code clair. Mais, en pratique, il est possible d'utiliser seulement le carré.

- ## Labyrinthe

Quatre types de labyrinthes ont été implémentés dans ce programme.

- Un labyrinthe « **Rectangle** » où une case est soit un mur soit un couloir. Ce labyrinthe est donc entouré d'une couche de mur sur tout son tour.

- Un labyrinthe « **Carré** » où chaque case est représentée par deux murs (gauche et haut), le mur bas est le mur haut de la case du dessous et le mur droit est le mur gauche de la case de droite.
- Un labyrinthe « **Hexagonal** » où chaque case est représentée par 3 murs (gauche, haut gauche, haut droit) et les murs restants correspondent aux murs de cases adjacentes à l'image du labyrinthe carré.
- Un labyrinthe « **Circulaire** » où l'on utilise une matrice carrée comme carte où le cercle s'enroule autour de la première ligne qui ne contient pas de mur. Une structure en forme d'arbre aurait été un autre choix possible mais par manque de temps, une implémentation avec une matrice carrée est une bonne alternative.

```
typedef struct Lab{
    Matrice * map;
}Labyrinthe;
```

```
typedef struct LabCarre{

    MatriceCarre * map;

}LabyrintheCarre;
```

```
typedef struct LabHexa{

    MatriceHexa * map;

}LabyrintheHexa;
```

```
typedef struct LabCercle{

    MatriceCarre * map;

}LabyrintheCercle;
```

Remarque : En effet, les structures carrées et cercles sont identiques mais elles le sont volontairement pour faire la différence entre les éléments en paramètre de fonction.

• Heap

Un tas binaire (min heap) a permis d'optimiser certains algorithmes en permettant une insertion triée d'éléments, et de maintenir le tas trié lors de l'extraction d'un élément.

Ici le tas est une simple table d'éléments triés par rapport à l'heuristique de l'élément.

```
/* Noeud de l'ensemble */
typedef struct{
    /* data*/
    int x; // [x,y]
    int y;
    int heuristique;
    int cout;

}Data;

typedef struct heap
{
        unsigned int size; // Size of the allocated memory (in number of items)
        unsigned int count; // Count of the elements in the heap
        Data ** data; // Array with the elements
}Heap;
```

Toutes les structures sont accompagnées d'accesseur et de fonctions qui leur sont propres.

Algorithme de génération

• Labyrinthe Rectangle

Le labyrinthe rectangle utilise une génération par grainage où quelques murs sont placés aléatoirement sur la grille et où les graines s'étendent comme des arbres.

Cette génération donne des labyrinthes parfaits avec plusieurs chemins possibles de l'entrée (case en haut à gauche) jusqu'à la sortie (case en bas a droite).

Le nombre de chemins dépendra du nombre de graines.

• Labyrinthe Carré

Le labyrinthe carré peut être généré de deux manières à partir d'un algorithme de Prim ou par fusion aléatoire de chemin (algorithme de Kruskal). Ces deux algorithmes donnent des labyrinthes parfaits avec des chemins uniques et un certain de nombre de voies sans issue.

Nous avons préféré coder un algorithme de Prim plutôt que l'exploration exhaustive récursive car l'algorithme proposé réalise des labyrinthes avec de longs couloirs mais peu de chemin sans issue, ce qui est moins intéressant à résoudre. De plus, l'algorithme n'est pas extrêmement compliqué et on a préféré réaliser un algorithme plus long et plus visuel.

Pour la fusion aléatoire de chemins, nous avons choisi de donner des valeurs aux cases et regrouper des cases en supprimant un mur entre les zones qui n'ont pas la même valeur puis de changer toutes les cases qui ont la valeur 2 par la valeur 1. La dernière étape augmente en effet la complexité de l'algorithme mais c'est une opération nécessaire si nous attribuons une liste de case dans la même zone à chaque case.

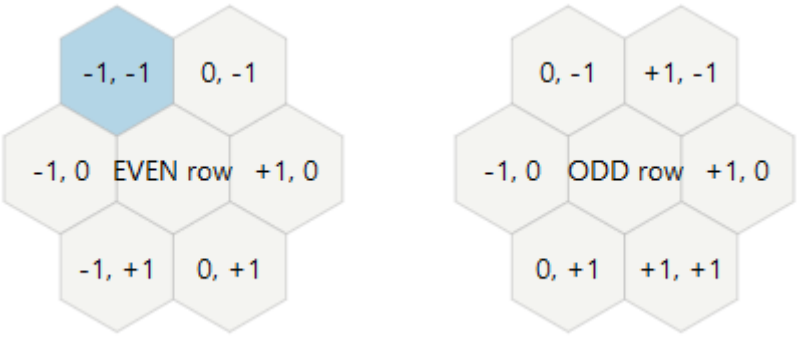
Cette solution est donc assez simple à implémenter bien que l'algorithme soit assez gourmand (lent).

• Labyrinthe Hexagonal

Pour la génération du labyrinthe hexagonal, nous avons aussi utilisé l'algorithme de Prim adapté au type de case. Cet algorithme donne un labyrinthe parfait avec des chemins uniques et de nombreux chemins sans issue.

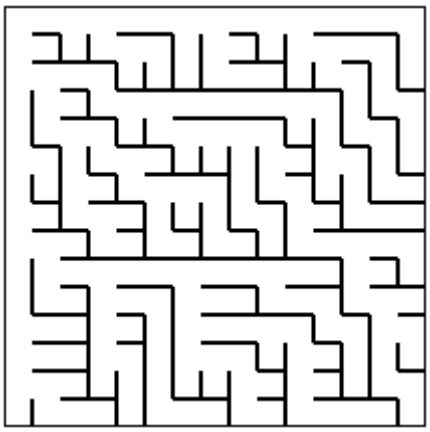
L'implémentation de labyrinthe hexagonal utilise le système de coordonnée par Offset (2 coordonnées) sur une grille de type : odd-r.

Le système d'offset implique que les coordonnées des cases adjacentes dépendent de la ligne où l'on se trouve. Elles ne seront pas identiques que l'on soit sur une ligne paire ou impaire.



- Labyrinthe circulaire

Pour ce type de labyrinthe, on a utilisé une variante de l'algorithme de génération par arbre binaire où le labyrinthe ressemble à un arbre binaire affiché en carré.



Une première ligne vide qui représente le centre du labyrinthe et chaque étage en dessous est un nouveau cercle.

Cet algorithme est simple et assez rapide (O(N), N = nombre de case). En effet, il suffit de parcourir chaque ligne une seule fois et sans comparaisons ou contrôle, on supprime soit le mur gauche soit le mur haut (sauf sur la première ligne où on supprime seulement les murs gauches).

Avec cet algorithme on évite d'obtenir le chemin vertical (à gauche de l'image ci-dessus) mais de ce fait il y a une faible chance qu'une ligne n'ait que des murs gauches supprimés et que le labyrinthe ne soit donc pas parfait.

Une simple modification qui vérifie le nombre de portes vers le haut sur une ligne (avec un compteur) nous permet de savoir si l'on doit casser un mur (aléatoirement) ou non

afin de créer un chemin. Pour rendre ce labyrinthe circulaire il suffit de l'afficher en enroulant en cercle autour de la première ligne.

Algorithme de recherche

Nous avons choisi de ne pas coder les algorithmes proposés, on a préféré implémenter des algorithmes plus efficaces et plus intéressants mais également plus complexes.

Ainsi, pour chaque labyrinthe nous avons implémenté l'algorithme de Dijkstra et l'algorithme A*.

Le premier, plus gourmand, nous retourne **le** chemin le plus court ; ce qui est particulièrement intéressant dans le cas d'un labyrinthe ou d'une simple carte avec plusieurs chemins.

Le second est l'algorithme le plus rapide en termes de recherche de chemin mais qui ne retourne pas forcement le chemin le plus court.

Il est aussi possible de faire une résolution manuelle en dépassant un robot manuellement, avec les flèches du clavier.

Remarque : La recherche de chemins n'est pas implémentée pour le labyrinthe en cercle, ce qui est dû aux limitations d'affichage aux alentours du centre l'implémentation. Elle est (quasiment) identique à celle du labyrinthe avec murs carrés à l'exception de la condition d'arrêt qui est une ligne et non une case.

Les différentes implémentations ont bénéficié d'une optimisation propre à l'implémentation des labyrinthes.

- ### Dijkstra

La recherche de la prochaine case est la partie de l'algorithme la plus complexe, étant donné que la distance entre deux cases n'est que de 1. En utilisant nos listes chainées (Ensemble) comme une file de case (First In – First Out) on obtient une complexité de recherche linéaire en O(1). Cette optimisation permet donc de rendre l'algorithme bien plus rapide.

- ### A *

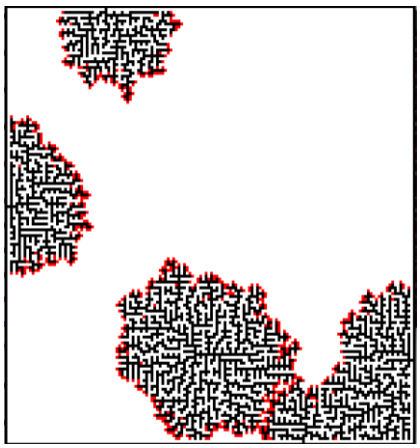
Dans cet algorithme, c'est aussi la recherche de la prochaine case à visiter ; ici avec la plus petite distance du point d'arrivé, distance de Manhattan (l'heuristique) ; qui est l'opération la plus gourmande. Mais l'implémentation d'un Heap (min heap) permet de gagner énormément de temps dans le processus de recherche qui devient d'une complexité en O(1),

légèrement au détriment de l'insertion de la case dans le Heap dû au tri effectué. Cependant, ceci demeure toujours plus rapide que d'effectuer un tri d'une liste à chaque insertion.

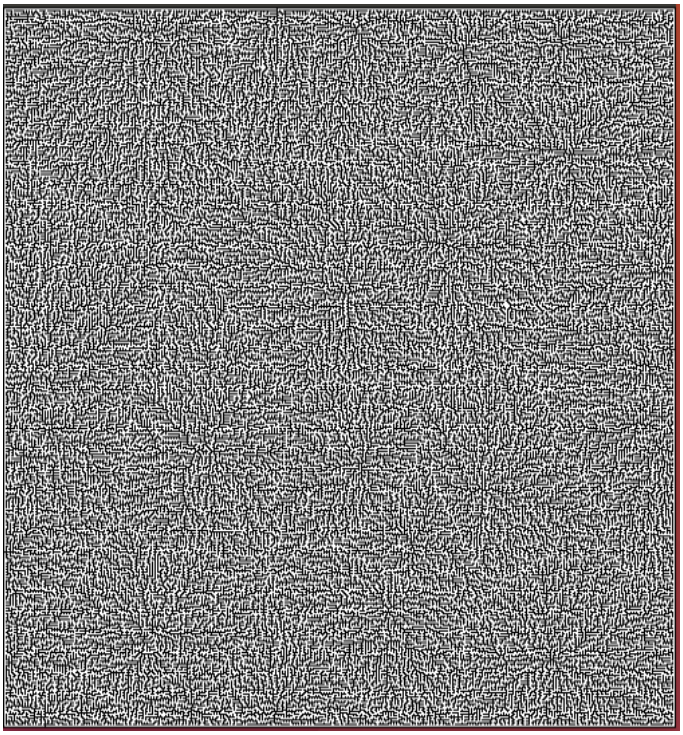
A noter que dans le cas du labyrinthe hexagonal, le calcul de l'heuristique était particulièrement compliqué par la nécessité de convertir l'offset en coordonnées cubiques (3D) pour le calcul.

Résultat

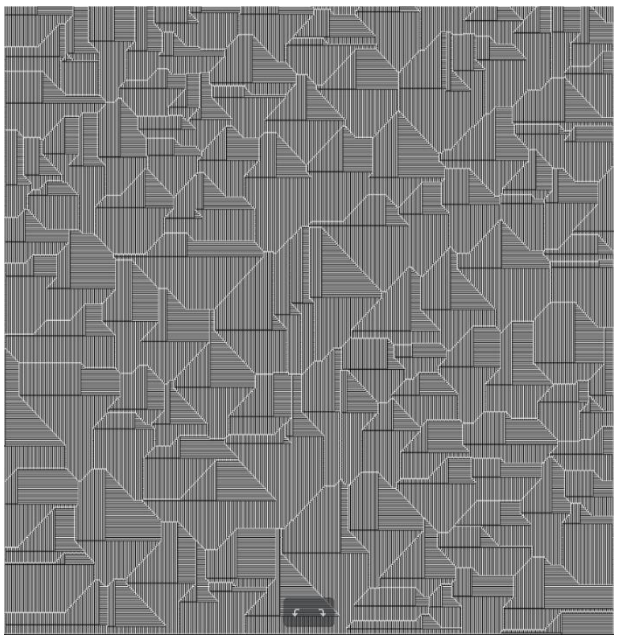
- Labyrinthe rectangle



Génération détaillée



Labyrinthe géant (ressemblant à une forêt)



Génération linéaire

```
./program -r -h 400 -l 400 -g 0.00001 -D -A
Génération
 temps d'execution: 0.22287 secondes

Dijsktra
 longueur du chemin : 988
 temps d'execution: 0.01143 secondes

A*
 longueur du chemin: 1012
 temps d'execution: 0.01711 secondes
```

Résultats

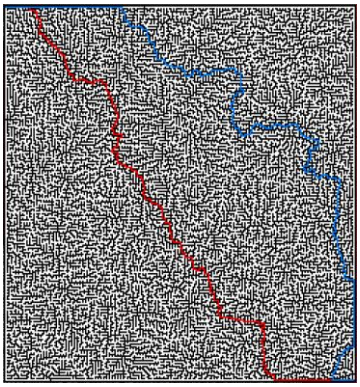
Les résultats de génération instantanée sur des labyrinthes moyens restent convenables pour de très grands labyrinthes. (2000*2000 => 4 000 000 cases)

Les algorithmes de recherche optimisés nous permettent d'obtenir un résultat de manière instantanée. Comme prévu, A* est (en général) plus rapide mais obtient un chemin plus long que Dijkstra. Mais il peut aussi être moins efficace en fonction de la configuration comme dans le test 1 où il se ''perd''.

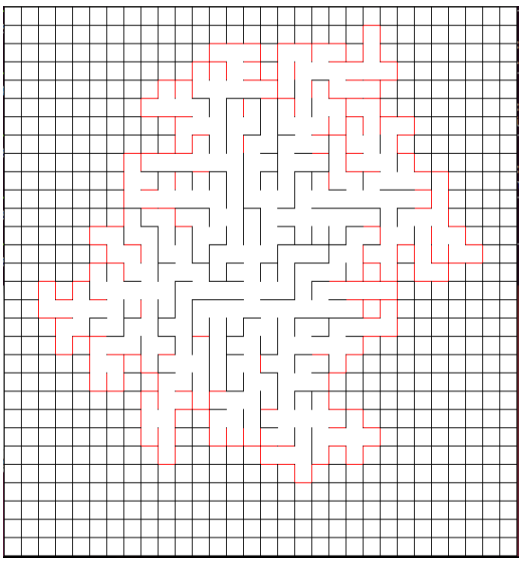
```
 ./program -r -h 1000 -l 1000 -g 0.00001 -D -A
Génération
 temps d'execution: 8.67305 secondes

Dijsktra
 longuleur du chemin : 2256
 temps d'execution: 0.15731 secondes

A*
 longueur du chemin: 3104
 temps d'execution: 0.02842 secondes
```

```
 ./program -r -h 2000 -l 2000 -g 0.0000001 -D -A
Génération
 temps d'execution: 35.85649 secondes

Dijsktra
 longuleur du chemin : 5170
 temps d'execution: 0.41226 secondes

A*
 longueur du chemin: 5172
 temps d'execution: 0.12348 secondes
```

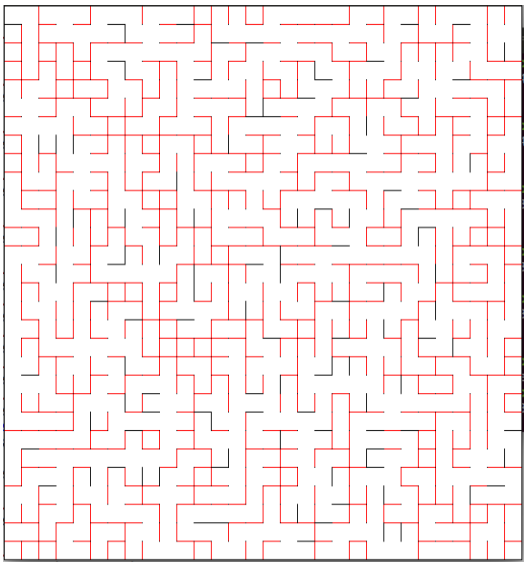
En rouge, Dijkstra, plus court et en bleu A* :



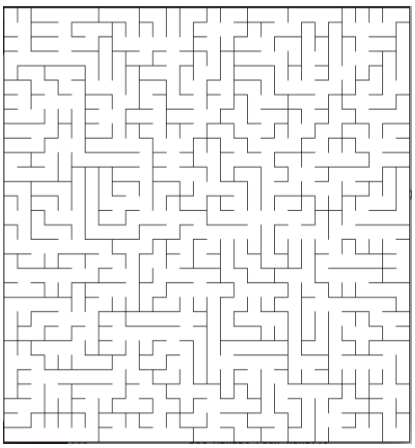
- Labyrinthe carré



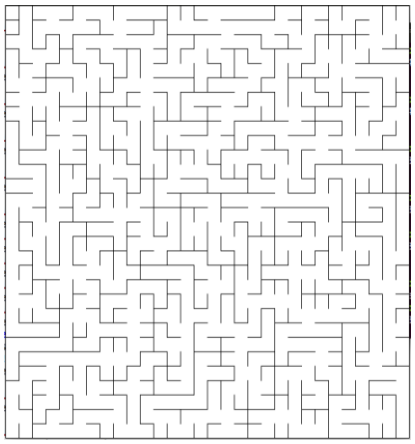
Génération de Prim



Génération de Kruskal (fusion)



Résultat par Prim



Résultat par Kruskal

```
 ./program -c -h 400 -l 400 -D -A -making 1
Génération
 temps d'execution: 2.14285 secondes

Dijsktra
 longueur du chemin : 856
 temps d'execution: 0.11190 secondes

A*
 longueur du chemin: 856
 temps d'execution: 0.04226 secondes
```

Statistique Prim

```
 ./program -c -h 200 -l 200 -D -A -making 2
Génération
 temps d'execution: 28.86320 secondes

Dijsktra
 longueur du chemin : 964
 temps d'execution: 0.01166 secondes

A*
 longueur du chemin: 964
 temps d'execution: 0.01276 secondes
```

Statistique Kruskal

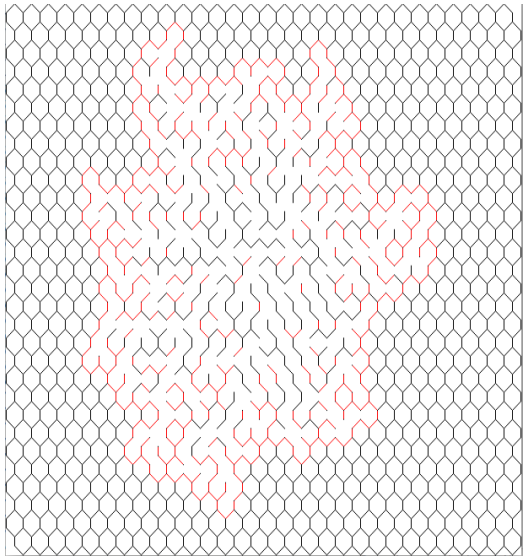
Sur ce labyrinthe, l'algorithme de Prim est assez rapide sur des labyrinthes moyens mais risque d'être beaucoup plus long. Ceci est dû à la complexité de notre implémentation de l'algorithme par fusion : $O(N^3)$.

Pour 200 * 200 le nombre de contrôle s'élève donc à 1 600 000 000.

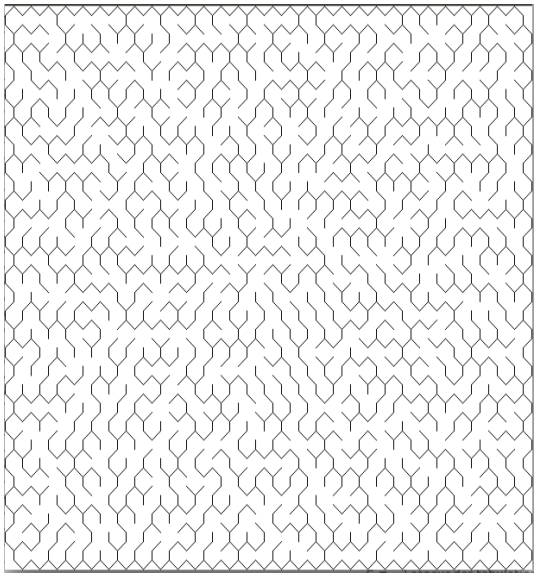
Les algorithmes de recherche ici aussi sont instantanés.

- Labyrinthe héxagonal

Génération par Prim



Résultat de la génération

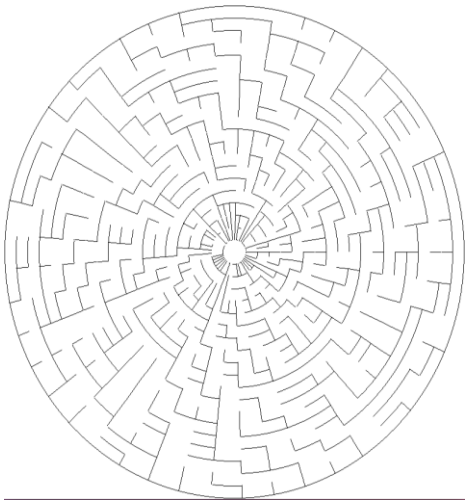


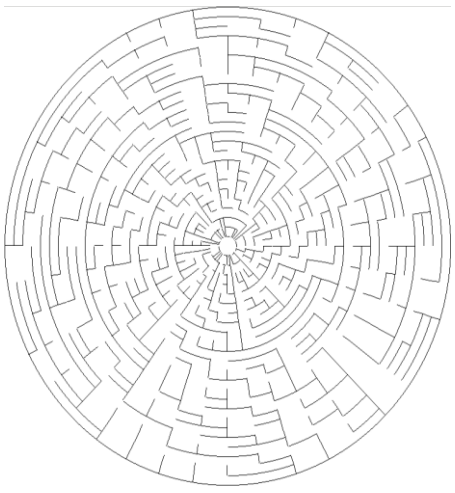
Statistique par Prim

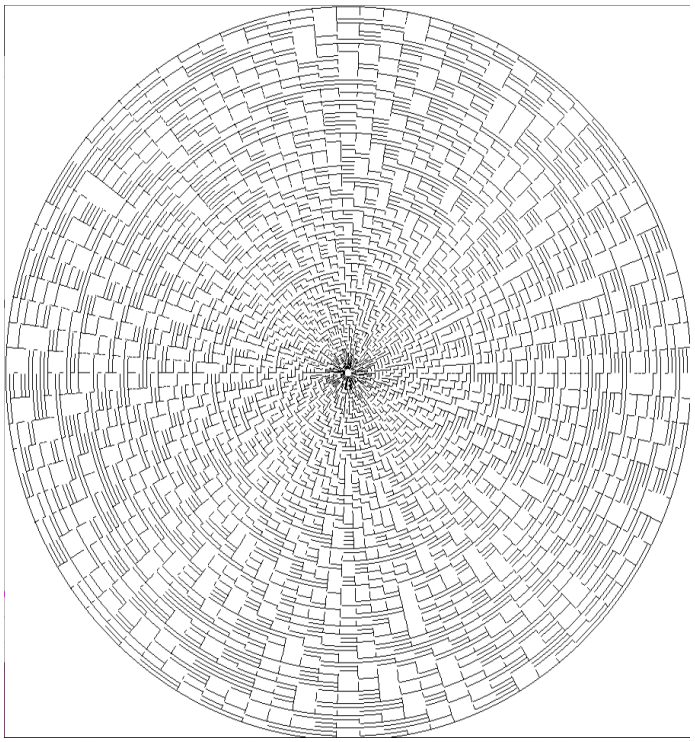
```
 ./program -x -h 400 -l 400 -D -A
Génération
 temps d'execution: 13.61003 secondes

Dijsktra
 longueur du chemin : 820
 temps d'execution: 0.12887 secondes

A*
 longueur du chemin: 820
 temps d'execution: 0.07735 secondes
```

Ici, comparés à ceux du labyrinthe carré, les temps sont plus importants. Ceci est dû au nombre de côtés de chaque case qui augmente la taille des ensembles plus rapidement et que rend la recherche des éléments plus longue.

- Labyrinthe circulaire







```
 ./program -C -h 400 -l 360
Génération
 temps d'execution: 0.00783 secondes
geoffrey@geoffrey:~/Documents/Labirynth
 ./program -C -h 1000 -l 360
Génération
 temps d'execution: 0.03662 secondes
```

Ici, l'algorithme étant en O(N) sans comparaisons, la génération est extrêmement rapide.