UNIVERSITE CADI AYYAD
ECOLE NATIONALE DES SCIENCES APPLIQUEES
MARRAKECH

Université Cadi Ayyad

Rapport final

Projet de Labyrinthe

Realise par :

BOUIGHERDAINE Said
ICHOU Karima
OUFRID Yousra
SELLOUK Hind
TAHIRI Mouad
KASRY Hamza

Membres du jury:

Mme.ZRIKEM Maria
Mr.ELMARZOUQI Nabil
Mr.KARKOUCH Aimad

ENCADRE PAR:

Mr.KARKOUCH Aimad

Sommaire

Introduction:

Objectifs :

Créer un jeu de labyrinthes, et pour le faire :

- Représentation des labyrinthes avec des graphes
- Utiliser des algorithmes de génération.
- Utiliser des algorithmes de résolution.
- Utiliser une bibliothèque graphique

Historique :

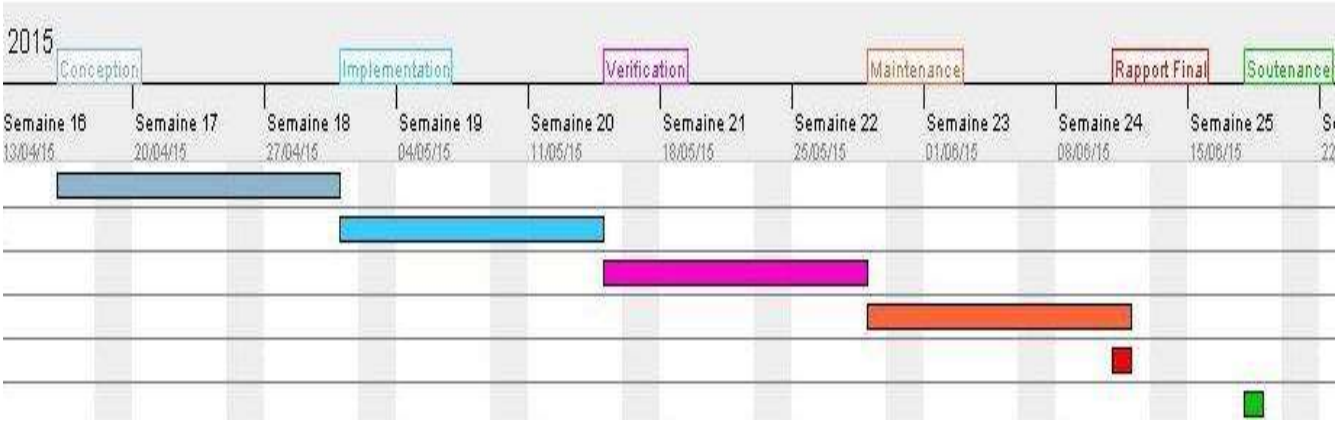
Un **labyrinthe**, est un tracé sinueux, muni ou non d'embranchements, d'impasses et de fausses pistes, destiné à perdre ou à ralentir celui qui cherche à s'y déplacer, leurs création fut depuis la préhistoire.

De nos jours, le terme de labyrinthe désigne une organisation complexe, tortueuse, concrète (architecture, urbanisme, jardins, paysages...) ou abstraite (structures, façons de penser...), où la personne peut se perdre. Le cheminement du labyrinthe est difficile à suivre et à saisir dans sa globalité.

Le 1er jeu de labyrinthes a été créé en 1986, par KOBBERT,et depuis lors plusieurs versions du jeu étaient apparu.

Planning :

Planning initial :



Planning effectif:

Etude fonctionnelle :

Les modules fonctionnels :

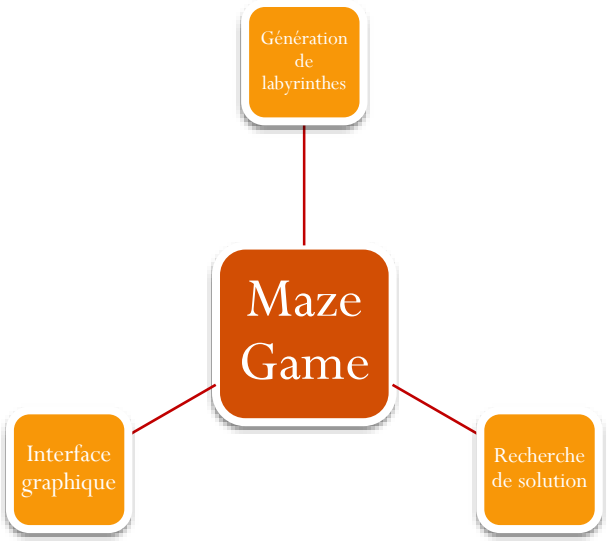
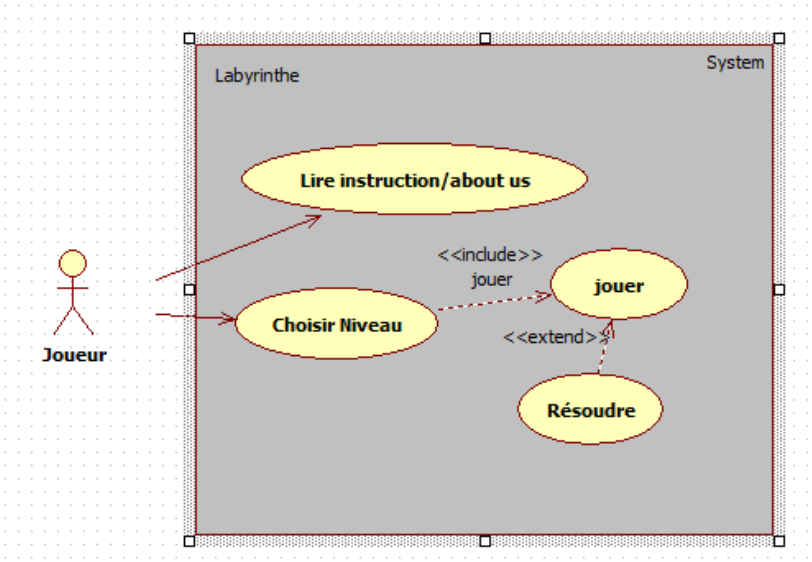
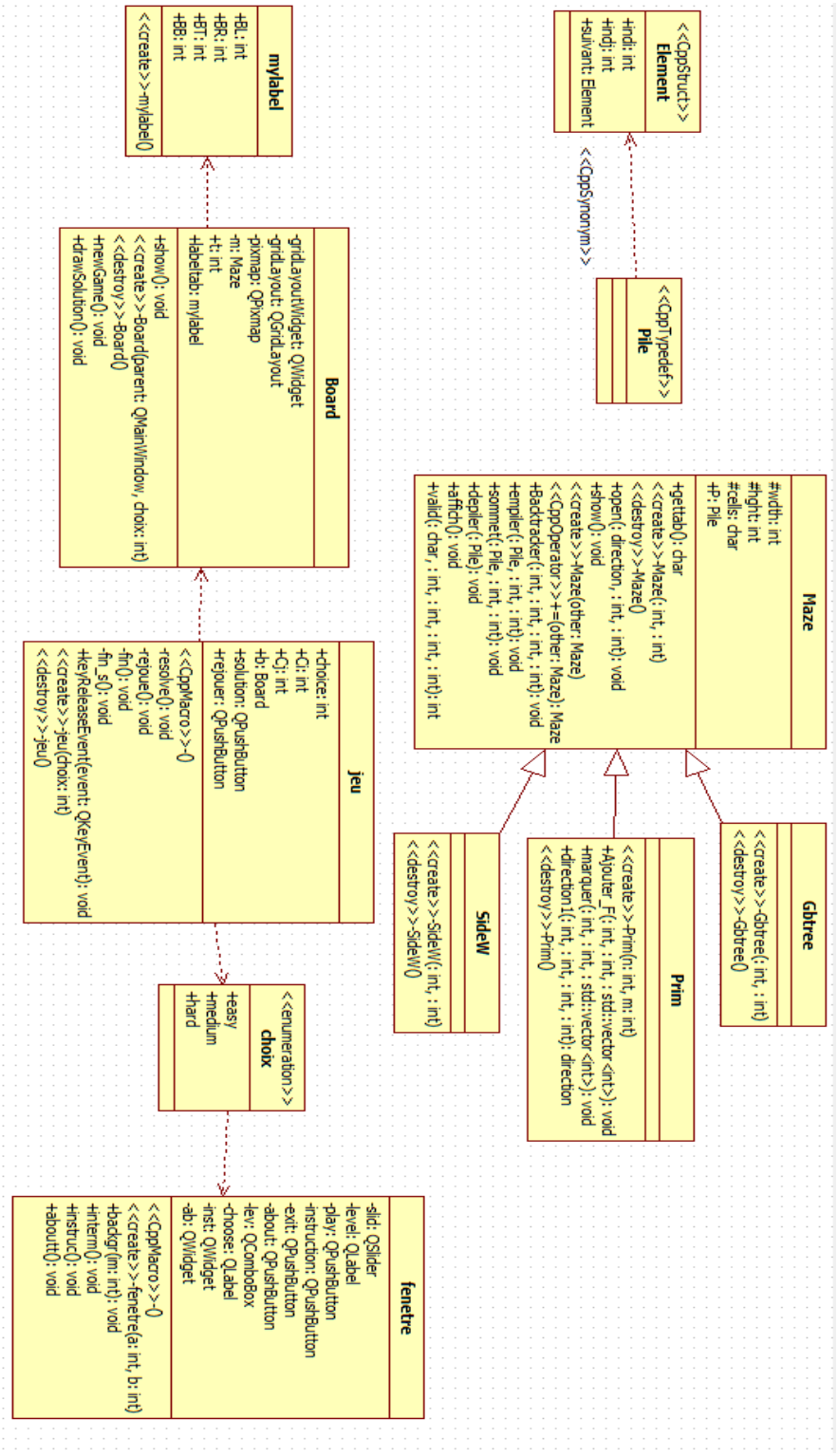


Diagramme de cas d'utilisation :



Les différentes classes utilisées :



Étude technique :

On ce qui concerne l'étude technique nous allons utiliser:

- Les bibliothéques de C++.
- Le développement va être effectué sous Windows, avec l'IDE CodeBlocks, QTcreator.
- Photoshop, QtDesigner, StarUML, GanttProject .

État de l'art:

Types de labyrinthes:

Les Labyrinthes sont classifiés selon plusieurs critères :

Dimension :

*2D:*La majorité des labyrinthes, sont soit sur papier, soit en taille réelle, et sont en cette dimension.
*3D:*Un labyrinthe avec plusieurs niveaux, où l'objet s'y circulant peut monter, descendre, et se diriger vers les quatre directions .

Hyper dimension(réfère à l'objet a déplacé) :

hypermaze: Où l'objet s'y déplaçant et plus qu'un point,ne peut exister qu'on 3 dimensions ou plus.
hyperhypermaze: Les dimensions de l'objet est augmentée, et peut être un plan.
Non hypermaze: Lorsque l'objet avec lequel on travaille est soit un point ou un petit objet.

Topologie:

Planaire: Les ensembles des labyrinthes ayant des structures topologiques anormaux, tel sur une surface d'un cube.
Normal: Un labyrinthe standard dans un espace euclidien.

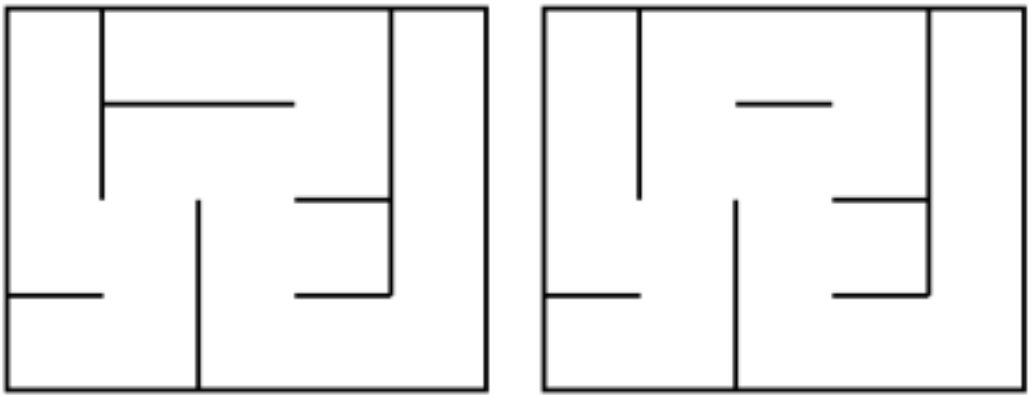
Les routes:

Parfait: Un Labyrinthe dans lequel il n'y a pas de circuits et de zones inaccessibles.
Braid: Un labyrinthe dans lequel on ne trouve pas d'impasses.

Les algorithmes de génération:

On peut distinguer, selon la méthode de génération, entre deux types de labyrinthes:
Labyrinthes parfaits : correspondant à un seul chemin passant par toutes les pièces.
Labyrinthes imparfait : offrant plusieurs chemins pour la sortie de labyrinthes.
Dans notre projet, on s'intéressera uniquement aux labyrinthes parfaits.

Un labyrinthe rectangulaire parfait de *m* colonnes par *n* lignes est un ensemble de *m* x *n* cellules reliées les unes aux autres par un chemin unique.



labyrinthe parfait

labyrinthe imparfait

On peut distinguer entre plusieurs algorithmes permettant de générer des labyrinthes, suivant leurs complexités, temps d'exécution, et niveau de difficulté (génération des impasses et de chemins diagonaux).
On peut citer, parmi les plus utilisés :

Algorithme de l'arbre binaire :

Il est simple (les labyrinthes ont tendances à avoir des taches : long couloir couvrant 2 cotes , routes en diagonale) et le principe consiste à choisir aléatoirement , pour chaque cellule , soit un de passage vers le nord ou vers l'ouest .
Cet algorithme peut être utile pour génération du labyrinthe avec moins d'impasse (niveau facile).

Algorithme de sidewinder :

Repose sur le même principe que celui des arbres binaire (et plus rapide) ; à la place d'ouverture par cellule, cet algorithme effectue une ouverture du nord au niveau d'une combinaison de cellules adjacentes.

Inconvénients :

Facile à parcourir du bas vers le haut : chemin vertical avec des petites déviations.

Algorithme de backtracker :

C'est un bon algorithme pour la résolution, mais peut aussi générer de bons labyrinthes.
Son principe porte sur l'ajout de chaque nœud visité dans une pile , et une fois se trouvant devant une impasse : dépile les nœuds visites jusqu'à son arrivé à un nœud qui présente une possibilité d'un autre chemin .

inconvénient :

Peu d'impasse et les chemins sont diagonaux.

Algorithme de KRUSKAL :

Cet algorithme est utilisé pour la génération des arbres couvrant minimaux. Il consiste à fusionner au hasard des nœuds adjacents en des morceaux, jusqu'à couvrir tous les nœuds. Cependant, il n'est autorisé à fusionner les nœuds qui ne font pas déjà partie du même morceau.

Algorithme de PRIM :

C'est aussi un algorithme de génération d'arbre couvrant minimale, Cependant, si nous remplaçons les poids des nœuds par des valeurs aléatoires, nous obtenons un générateur d'arbre couvrant aléatoire!
le principe est : choisir un nœud de démarrage aléatoire , l'ajouter à l'arbre et marquer tous ses voisins comme frontières , choisir une des frontières et la marquer avec un voisins déjà visité et marquer tous ses frontières , ainsi de suite jusqu'à ne plus avoir des nœuds de frontière .

Algorithme d'arbre en croissance :

Son principe est de choisir un nœud aléatoirement et l'ajouter à un "champ actif", ajouter un de ses voisins non visités au champ, et au cas où le nœud n'a pas de voisins : le supprimer du champ, ainsi de suite, jusqu'à vider tout le champ.
La particularité de cet algorithme est que selon la méthode de suppression de nœud, on peut tomber sur deux algorithmes distincts :

- le plus nouveau : si on choisit le dernier nœud ajouté on retombe sur l'algorithme de backtracker.
- au hasard : si on choisit un nœud au hasard, on retrouve l'algorithme de PRIM.

Algorithme des arbres en croissance binaire :

Son principe est le même que celui des arbres en croissance, la différence est qu'il ajoute deux nœuds ou plus au champ actif (il est donc plus rapide). Ainsi il permet la génération de grands labyrinthes plus complexes, et avec plus d'impasse (niveau difficile) .

Les algorithmes de résolution :

Il existe plusieurs algorithmes de résolution qu'on peut classer suivant plusieurs critères:

>Consommation en temps
>Consommation en mémoire
>Nombre de solutions trouvées
>La meilleure solution (chemin le plus court)

Ps: pour nous on s'intéresse au labyrinthe parfait où il n'existe qu'une seule solution
Parmi ces algorithmes on trouve:

Algorithme du backtracking:

Rapide, marche sur tous les types de labyrinthes, il trouve une seule solution (pas forcément la meilleur en cas de labyrinthes imparfaits).Mais nécessite une pile dont la taille peut atteindre au pire cas le nombre de cellule du labyrinthe donc sa complexité est en O(n) où n est le nombre de cellules.

Fonctionnement:

Si l'on rencontre une impasse ou une cellule inexplorable (cellule condamnée ou déjà explorée) on va remonter jusqu'à l'intersection précédente (en dépilant). Chaque cellule dépilée est marquée

Sinon, si on a atteint la sortie alors l'algorithme se termine ;

Sinon on se déplace dans une des directions libres.

Dead-end filling algorithme:

Contrairement au Backtracking cet algorithme n'a pas besoin d'une mémoire supplémentaire.
Il cherche à trouver toutes les solutions possibles donc il est moins rapide que l'algorithme précèdent.

Fonctionnement:

Il parcourt le labyrinthe et à chaque fois qu'il rencontre une impasse il la remonte jusqu'à arriver à une intersection. Lors de cette remontée, toutes les cellules traversées sont marqué. À la fin, il ne restera que la (cas d'un labyrinthe parfait) ou les solutions.

Désavantage:

Dead-end filling doit parcourir entièrement le labyrinthe pour être sûr d'avoir traité toutes les impasses Or dans notre cas nous n'avons besoin que d'une seule solution puisque les labyrinthes parfait contiennent une unique solution.
Cela implique que cet algorithme va consommer plus de temps que nécessaire pour notre cas.

Wall follower:

C'est un algorithme simple de résolution, il est rapide car il n'utilise pas de mémoire supplémentaire.
Pour Chercher un passage, là où il y a un mur il tourne à droite (à gauche), ce qui est équivalent à la résolution humaine en mettant la main sur le mur droit (gauche), on peut marquer les cellules visitées. A la fin on choisit la solution en suivant le trajet où on a des cellules visités une seule fois.

Inconvenient :

- Avec cette méthode, on ne trouverait pas nécessairement la solution la plus courte.
- Il ne fonctionne pas si l'objectif est dans le centre du labyrinthe et il y a un circuit fermé qui l'entoure car finalement on se retrouve au début.

La structure de données :

Structure : Bitwise

Les cellules contiennent, chacune, 4 portes, et chaque porte n'a que deux états : fermé ou ouvert . On aura donc besoin que de 4 bits pour l'état des murs plus un bit pour savoir si une cellule est déjà visitée.
Ainsi le tableau de cellules peut se représenté par un vecteur de m*n cases tout en permettant de ne stocker qu'une seule coordonnée afin de faciliter l'adressage d'une cellule.
donc :

Une cellule sera codée sur 1 char .

Nord sera le bit 0, Ouest le 1, Sud le 2, et Est le 3.
L'état, visité ou non sera codé sur le bit 4.

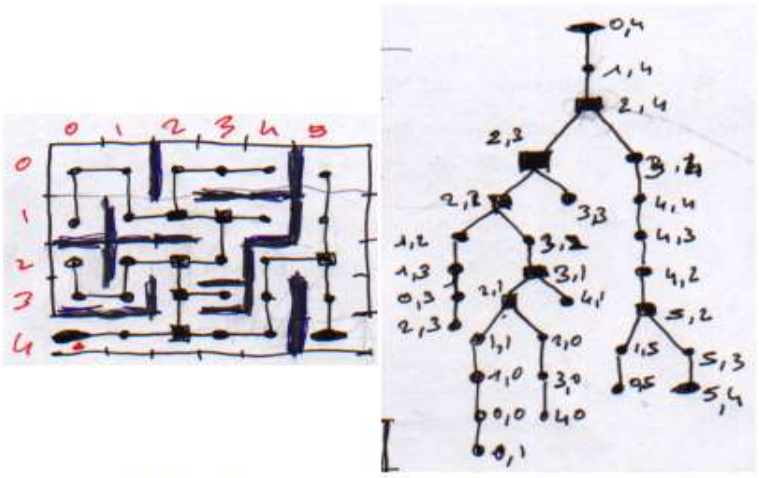
Le labyrinthe contiendra ainsi 2 entiers m et n, les dimensions du labyrinthe. L'entrée et la sortie seront codes par des pointeurs, plus le tableau de caractères qui contiendra les cellules.
Voici la structure de données -Bitwise- :

```
typedef struct maze maze;
struct maze
{
    int hght;
    intwdth;
    char* in,out;
    char** Laby;
};
```

Représentation par un graphe :

```
typedef struct CELL CELL;
typedef struct NODE NODE;
struct NODE{
            int i;
            int j;
            CELL *adjacent;
            };
struct CELL{
            CELL* next;
            NODE son;
};
```

Le graphe représentant les chemins de labyrinthe sera représenté par une liste chaînée : chaque cellules pointe vers ses successeurs (une cellule ne peut pas avoir plus de deux successeurs pour un labyrinthe parfait).



Labyrinthe Arbre équivalent

Conception:

Les algorithmes de génération:

Le SideWinder:

Principe:

Consiste à choisir aléatoirement, pour chaque cellule, un passage à ouvrir soit vers le nord soit vers l'est.
Quand on n'a pas ouvert vers le Nord un ensemble de cellules adjacentes, cet algorithme effectue une ouverture du nord au niveau d'une de ces cellules aléatoirement.
Génère des labyrinthes faciles à parcourir du bas vers le haut : chemin vertical avec des petites déviations.

Pseudo-code:

```
   1. Le parcours de la grille se fait ligne par ligne, en
      commençant par la première la première cellule de la
      première ligne. On initialise le champ à utiliser.
   2. Ajouter la cellule courante dans le champ.
   3. Décider de la direction de création de passage.
   4. Si le nouveau passage a été créé nouvelle cellule deviendra
      la cellule courante, et refaire les étapes de 2-4.
   5. Si le passage n'a pas été créé, choisir n'importe qu'elle
      cellule dans le champ (de cellules).
      - créé un passage de direction du Nord
      -Vider le champ.
      Faire de la cellule suivante de dans la ligne, la cellule
courante, et refaire 1-5.
   6. Continuer jusqu'à ce que toutes les cellules soit traiter.
```

Algorithme de PRIM :

Principe:

C'est un algorithme de génération d'arbre couvrant minimale, Cependant, si nous remplaçons les poids des nœuds par des valeurs aléatoires, nous obtenons un générateur d'arbre couvrant aléatoire.

On aura besoin**,** d'une place mémoire équivalente à la taille du labyrinthe. Tout au long de la création, chaque cellule est dans l'un de ces 3 états :

Dedans : la cellule a déjà été creusé, un passage a été créer.

Frontière : la cellule n'est pas une partie du labyrinthe mais elle est adjacente à cellule de type (1).

Vierge: la cellule n'est ni « Dedans » ni « Frontière ».

Pseudo Code :

1. On prend une cellule « Vierge » aléatoirement, début de la fonction récursive
2. On la marque comme « Dedans ».
3. On liste ses voisines (Cellules « Frontière »).
4. Si les voisines ont déjà toutes été visitées :
    a. On recommence avec la case précédente
5. Sinon (si au moins une voisine n'a pas été visitée)
    a. On ne garde que les voisines non visitées
    b. On en choisit une au hasard.
    Si (le chemin n'est pas complet)
        On recommence avec la case choisie
    Sinon
        a. On enregistre le passage
        b. On marque cette cellule voisine comme visité et on l'ajoute au labyrinthe

Le labyrinthe est terminé lorsqu'il n'y a plus de cellules « Frontière » (ce qui signifie qu'il n'y a plus de cellules « Vierges », c-à-d qu'elles sont toutes « Dedans »).

Growing binary tree :

Principe:

On choisit jusqu'à deux nœuds voisins aléatoirement, on les ajoute au champ actif ajouter un de leurs voisins non visités au champ, et au cas où le nœud n'a pas de voisins : le supprimer du champ, et ressayer avec un autre, jusqu'à vider tout le champ.
(il est donc plus rapide), ainsi il permet la génération des grands labyrinthes plus complexes, et avec plus d'impasse.

Pseudo-code:

```
      méthode de backtracker (choisir le dernier nœud ajouté):
            1)  initialisation : pile S vide.
                        W (ensemble des sommet marqués définitivement) vide .
            2) boucle principale :
                  2.1) choisir le sommet d'entrée in , le marqué s  et
l'empiler dans  S :
      S={S}U{in}.
                  2.2) soit x1 le sommet de la pile
      2.3) marquer s un ou deux voisins de x1 .
      2.4) dépiler x1, empiler  les voisins marqués , et ajouter x1 a W
      2.5) si x1 n'a pas de voisins non encore marqués  le dépiler de S et
aller en
           2.2
      sinon, aller en 3 .
                  3) S =O , W=X fin de l'algorithme.
```

L'algorithme de résolution :

Backtracking :

Principe:

Rapide, marche sur tous les types de labyrinthes, il trouve une seule solution (pas forcément la meilleur en cas de labyrinthes imparfaits).Mais nécessite une pile dont la taille peut atteindre au pire cas le nombre de cellule du labyrinthe donc sa complexité est en O(n) où n est le nombre de cellules.

Pseudo-code:

```
      Soit "S" la case de départ et "END" la case  à
atteindre ,
      empiler "S" dans une pile "P"
      marquer "S"
      Tant que ("S" différent de "END")
      Si "S" à une porte ouverte vers une case "Y" non
marqué
      {
            marquer "Y";
            empiler "y" dans "P";
            S=Y;
      }
      Sinon
      {
            Depiler(P);
            S=sommet(P);
      }
```

Structure de donnée Adoptée :

```
typedef struct maze
maze;
struct maze
{
    int hght;
    int wdth;
    char* in,out;
   char** Laby;
};
```

On a choisi la structure de données -Bitwise- :

Pour le traitement des bits , on se basera sur les opérateurs binaires.

& l'opérateur AND

| L'opérateur OR

^ L'opérateur XOR

~ l'opérateur d'Inversion

>> opérateur de décalage à droite

<< opérateur de décalage à gauche.

Quelques fonctions utiles :

1. ouvrir les portes entre l'ancienne et la nouvelle cellule :

- une porte ouverte est à 1 :

cellule |= 1 << i avec cellule : un caractère (char) .

i représente la direction (N = 0, W = 1, S = 2,E= 3)

le 1 est : 00000001 , et avec l'utilisation de l'opérateur << on décale le 1 de i positions vers la gauche .

Pour la porte N: 00000001

Pour la porte W : 00000010

Pour la porte S : 00000100

Et pour la porte E: 00001000

Ensuite on applique le ou exclusif | pour changer l'état de ième bit dans cellule en 1 (porte ouverte).
Pour obtenir la porte équivalente dans la cellule suivante, il suffit l'ajouter 2 modulo 4 à i.

1. ouvrir les portes entre l'ancienne et la nouvelle cellule (pas besoin de cette fonction) :

- on procédera de la même façon sauf qu'on utilisera l'opérateur du ET exclusif :

  Cellule &= ~(1 << i)

- On prend un caractère à 1, on décale le 1 de i positions vers la gauche.
- on prend le complémentaire de ce nombre avec l'opérateur d'Inversion (pour garder juste le ième bit a 0).
- on effectue un ET logique avec la cellule

Réalisation :

Introduction:

Après avoir construit une idée claire sur le projet et ses exigences et comment les développées, et les différent algorithmes qu'on va utiliser, le présent chapitre décrit la phase de réalisation de l'application, on commence par :

Interface graphique :

Choix d'interface :

Pour l'interface graphique, on a décidés de travailler avec QT, tant qu'elle représente plus d'outils que les bibliothèques GTK+.

Qu'est ce que QT?

Qt est un **framework** multiplateforme pour créer des GUI (programme utilisant des fenêtres).
Qt est écrite en C++ et elle est, à la base, conçue pour être utilisée en C++

Qt est donc constituée d'un ensemble de bibliothèques, appelées « modules ». On peut y trouver entre autres ces fonctionnalités :

- **Module GUI** : c'est toute la partie création de fenêtres. Nous nous concentrerons surtout, dans ce cours, sur le module GUI.

- **Module OpenGL** : Qt peut ouvrir une fenêtre contenant de la 3D gérée par OpenGL.

- **Module de dessin** : pour tous ceux qui voudraient dessiner dans leur fenêtre (en 2D), le module de dessin est très complet !

- **Module réseau** : Qt fournit une batterie d'outils pour accéder au réseau, que ce soit pour créer un logiciel de Chat, un client FTP, un client Bittorent, un lecteur de flux RSS…

- **Module SVG** : Qt permet de créer des images et animations vectorielles, à la manière de Flash.

- **Module de script** : Qt prend en charge le Javascript (ou ECMAScript), que vous pouvez réutiliser dans vos applications pour ajouter des fonctionnalités, par exemple sous forme de plugins.

- **Module XML** : pour ceux qui connaissent le XML, c'est un moyen très pratique d'échanger des données à partir de fichiers structurés à l'aide de balises, comme le XHTML.

- **Module SQL** : permet d'accéder aux bases de données (MySQL, Oracle, PostgreSQL…).

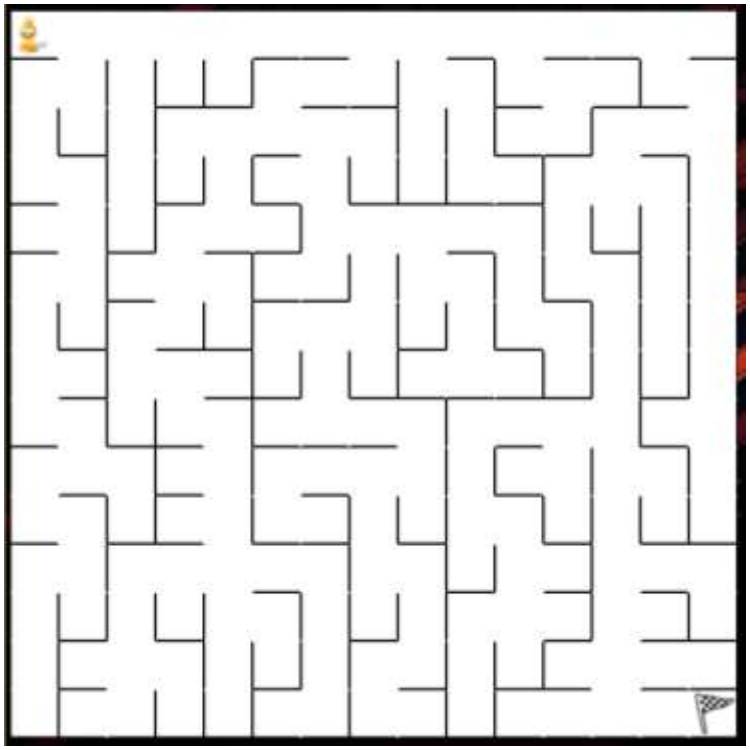
Les phases de construction:

1-Phase de génération graphique:

Les classes Utilisées:

```
class Board:public QWidget
{
      QWidget *gridLayoutWidget;
      QGridLayout *gridLayout;
      mylabel **labeltab;
};
```

Chaque cellule est représenté par un objet de type mylabel héritant de QLabel, les cellules se trouvent dans un objet de type QGridLayout :



2-Gestion des évènements et du temps :

Pour faire bouger le joueur on utilise les flèches du clavier.

Les évènements peuvent être reçus et pris en charge par n'importe quelle instance d'une sous-classe de **QObject**.

Pour gérer les événements il faut utilisé la fonction :

```
void keyReleaseEvent(QKeyEvent* event);
```

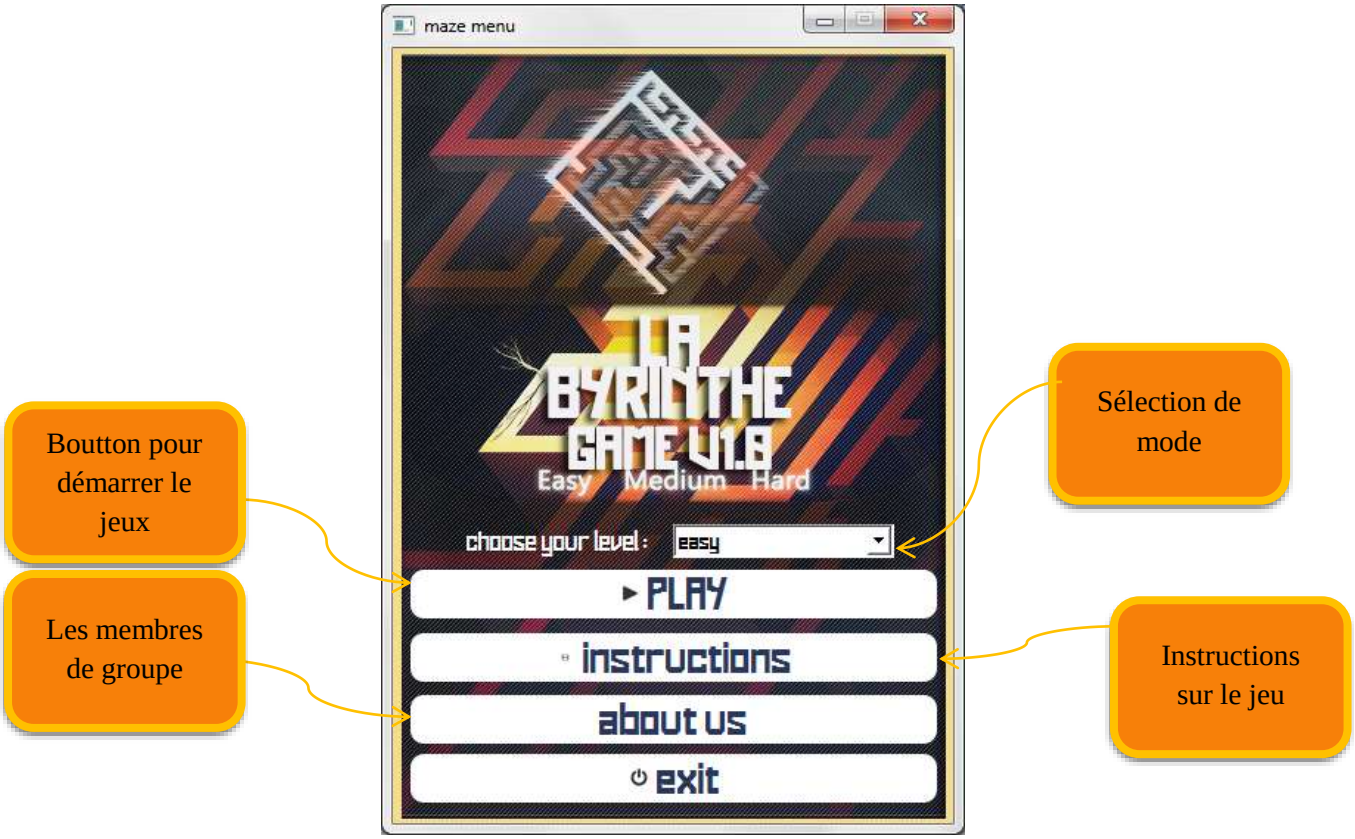
La classe QKeyEvent décrit un événement clavier elle contient la méthode key() qui prend les évènement :

```
Qt::Key_Right,QT::Key_Left,Qt::Key_Up,Qt::Key_Down.
```

En ce qui concerne le temps nous avons utilisé les classes : QTime et QTimer

3-Gestion des signaux et des slots:

Pour gérer les intersections entre le joueur (via l'interface GUI), on a eu besoin de créé et gérer plusieurs signaux et slots, tel les clicks sur le bouttons, et les choix dans les listes déroulantes.



Les classes utilisées :

QPushButton : Créer des Bouttons

Pour différents boutons tel : Play, instructions, about us, exit, rejouer, résoudre.

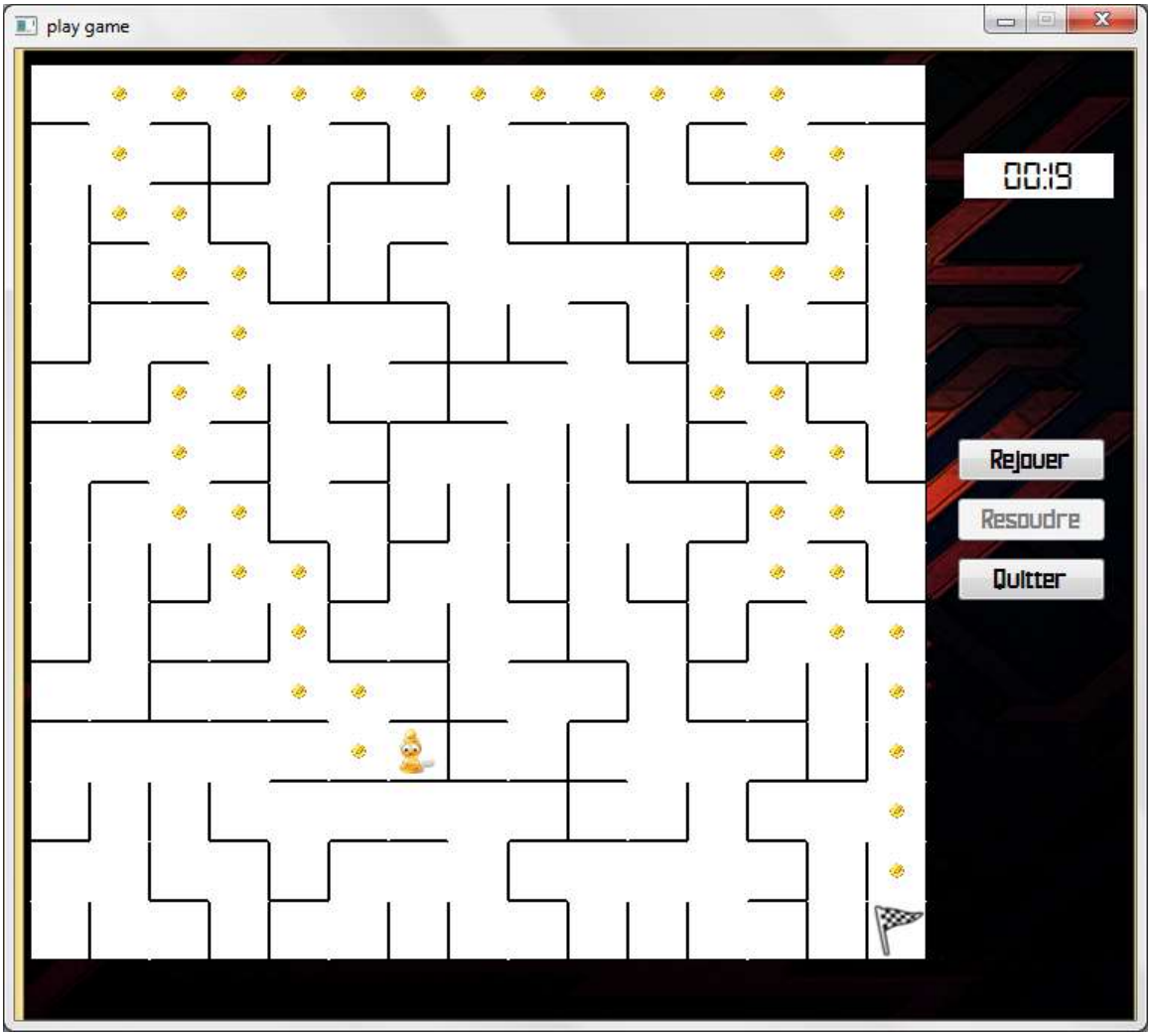
QComboBox : Liste déroulante pour choix de niveau.

QWidget: fenêtre de about us et fenêtre d'instructions

Quand le joueur atteint la destination une fenêtre de type " QMessageBox" s'affiche:

4-Phase de la résolution:

Quand l'utilisateur clic sur le boutton "résoudre" un signal est envoyé vers une fonction qui effectue la résolution grâce à l'algorithme "Backtracking" :



Conclusion et perspectives :

Le projet final respecte les principales fonctionnalités définit dans le cahier de charge du projet, et il est possible d'exécuter le programme et y jouer, et il est aussi possible de générer la solution.
Pour le planning, on n'a pas suivi le planning initial vue qu'on n'avait pas la phase de maintenance, et aussi vue l'importance critique des phases d'implémentation et de vérification.

Concernant les perspectives, il y a plusieurs améliorations qu'on peut apporter au projet, tel avoir un joueur qui se déplace, au lieu d'une image qui passe d'un label à un autre, augmenter la dimension et en faire un jeu 3D, ou encore augmenter les difficultés des labyrinthes produits.

Sources et bibliographie :

http://www.astrolog.org/labyrnth/algrithm.htm

http://fr.wikipedia.org/wiki/Labyrinthe_%28jeu%29

http://fr.wikipedia.org/wiki/Labyrinthe

http://www.jamisbuck.org/presentations/rubyconf2011/index.html#title-page