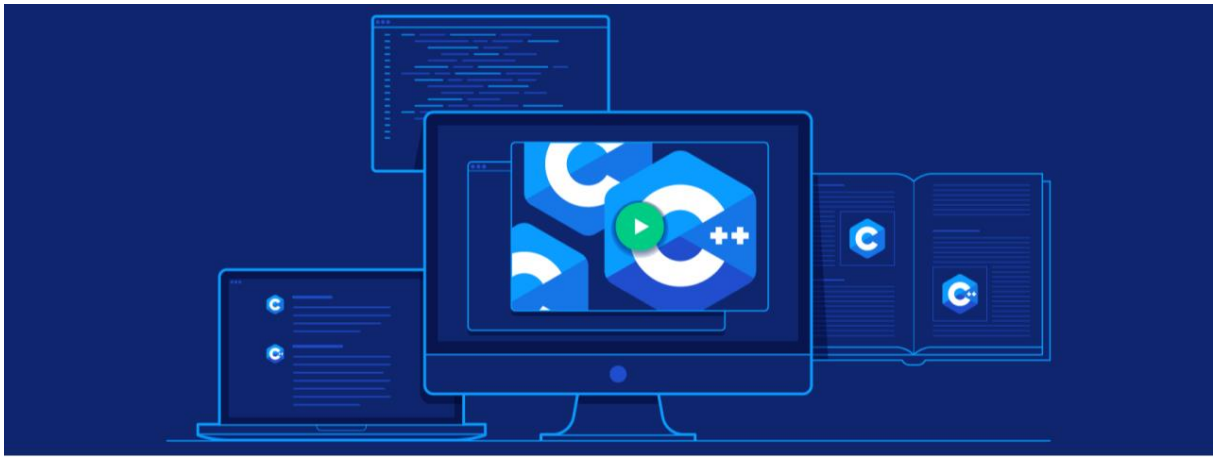
DOCUMENTATION DU PROJET LABYRINTHE

Année scolaire 2022-2023

POLYTECH NICE SOPHIA | UNIVERSITÉ CÔTE D'AZUR



BYMBOUTSA Charly Darcy

PIN Nolan

Table des matières

Cahier des charges

On souhaite programmer en C++ la création et la visualisation graphique d'un labyrinthe, duquel un utilisateur devra sortir en partant d'une position initiale vers une position finale de sortie. Pour l'interface graphique, vous utiliserez la bibliothèque Gtkmm.

L'utilisateur se déplacera dans le labyrinthe à l'aide des 4 flèches du clavier de l'ordinateur. Vous proposerez plusieurs labyrinthes de tailles variables construits automatiquement. La génération automatique des labyrinthes suivra obligatoirement deux algorithmes : Fusion et Aldous-Broder.

Pour ce projet nous avons utilisé comme modèle d'implémentation le modèle MVC.
L'implémentation du modèle Observateur / Observable n'a pas pu être faite.

Utilisation de l'application

Présentation

L'application se compose d'une seule fenêtre principale.

Celle-ci est composée de plusieurs partie :

- Un groupe de boutons en haut de la fenêtre qui concerne la résolution automatique du labyrinthe.
- Une partie centrale ou l'intégralité du labyrinthe et sa solution pourront être dessinées.
- Un groupe de bouton sous cette partie centrale. Ces boutons correspondent à la génération du labyrinthe avec les algorithmes associés.

Le labyrinthe a une taille par défaut de 3x3, les cases de départ et d'arrivée sont respectivement rouge, placée en haut à droite et verte, placée en bas à droite.

La position du joueur est représentée par un cercle blanc.

Fonctionnalité

Taille

La taille du labyrinthe peut être modifié a tout instant grâce aux deux entrées de textes intitulées *Largeur* et *Hauteur*. La validation de la nouvelle taille se fait soit en appuyant sur la touche Entrée du clavier en étant dans ces entrées de texte ou bien simplement en appuyant sur le bouton *Restart*. La taille ne peut pas être réglée en dessous de 2x2.

Déplacement

Le déplacement du joueur se fait uniquement via les touches fléchées du clavier. Le joueur est confiné dans les murs du labyrinthe et doit attendre la case d'arrivée verte situé en bas à droite.

Résolution automatique

Il y a trois boutons concernant la résolution du labyrinthe. Le bouton *resolver* affiche sur le labyrinthe la solution en bleu. Les boutons *Move auto* et *Move auto stop* on pour but de déplacer le joueur automatiquement le long de cette solution.

Problèmes

Après avoir sélectionné un algorithme de génération du labyrinthe ou tout autre bouton, ceux-ci restent par défaut sélectionnés. Le déplacement avec les touches fléchées ne modifie donc pas la position du joueur dans le labyrinthe mais modifie quel bouton est sélectionné. Pour remédier à cela, il faut systématiquement remonter vers la zone d'affichage avec les touches fléchées pour reprendre contrôle du joueur

Lorsque le mouvement automatique est activé, la position du joueur n'est pas miss a jour du coté du modèle. Si le joueur effectue un déplacement manuel en pendant le déplacement automatique, il y aura des problèmes de saut à l'affichage. Ce problème persiste tant que le déplacement automatique est en marche.

Modèle MCV

Le modèle MCV, Modèle Vue Contrôleur, permet de scinder les parties de code de façon très distincte. Il n'y a aucune interaction directe entre le Modèle et la Vue, toute liaison se fait à travers le contrôleur.

Architecture du modèle

La solution que nous avons mis en place est que le modèle est composé de cellule qui représente le labyrinthe qui est l'observable du modèle.

Dans les sections qui suivent et grâce au mot clé *using,* Le type *Coordinate* décrit une pair d'entier représentant les coordonnées dans le labyrinthe, celles-ci sont exprimées sous la forme : (x,y) avec x augmentant vers la droite et y augmentant vers le bas.

Cellule

Présentation générale

La classe *Cell* représente une cellule individuelle dans un labyrinthe. Elle contient des informations telles que les coordonnées de la cellule, son identifiant et les coordonnées de ses cellules voisines.

Elle permet de représenter et de manipuler les cellules du labyrinthe en maintenant une liste de voisins pour chaque cellule. Cette liste de voisin permet de déterminer les connexions entre les cellules, i.e. les ouvertures dans les murs, et de créer le labyrinthe de manière efficace.

Voisin

Chaque cellule du labyrinthe étant unique, il ne peut y avoir de doublons dans la liste des voisins (ou liste d'adjacence), c'est pourquoi le choix d'utiliser un **set<Coordinate>** est avantageux. En effet, la complexité d'accès aux valeur est en O(log n).

Labyrinthe

Présentation générale

La classe *Labyrinth* est une représentation d'un labyrinthe avec des fonctionnalités pour le créer, le manipuler et le résoudre. Elle utilise un graphe pour modéliser le labyrinthe, où chaque cellule est un nœud et les ouvertures dans les murs entre les cellules sont représentés par des arêtes.

Le labyrinthe peut être créé avec une taille spécifiée et rempli avec des cellules vides. Les murs entre les cellules peuvent être supprimés pour créer des chemins. Le joueur peut se déplacer à travers le labyrinthe en respectant les limites et les murs. Il peut également être réinitialisé à la position de départ. Différents algorithmes sont disponibles pour construire le labyrinthe.

L'algorithme de construction de mur *Fusion* supprime de manière aléatoire les murs entre les cellules jusqu'à ce que toutes les cellules soient connectées.

L'algorithme "Aldous-Broder" visite chaque cellule de manière aléatoire et supprime les murs entre les cellules visitées.

Le labyrinthe peut également être résolu en trouvant le chemin jusqu'à la case gagnante. L'algorithme de recherche en largeur (BFS) est utilisé pour trouver ce chemin.

Vérification

De nombreuses fonction dans cette classe permettent de faire interface avec le reste du programme.

L'intégralité de l'intelligence concernant le labyrinthe se fait en interne dans la classe et celle-ci renvoie simplement les résultats. Par exemple, la fonction *move_position* qui reçoit en paramètre une direction (0=ouest, 1=nord, 2=est, 3=sud), vérifie que le déplacement est conforme, i.e. pas vers un mur ou en dehors du labyrinthe, l'effectue puis renvoie une indication si le déplacement a été effectué ou pas.

Résolution

Un avantage majeur d'avoir mis en place un graphe pour représenter le labyrinthe est que la résolution se fait de façon très rapide. La résolution de chemin dans un graphe est un problème mathématique générique déjà résolu. Le *BFS*, Breadth-First Search ou algorithme de parcours en largeur, permet de trouver un chemin reliant deux points. Sachant qu'il n'existe qu'une seule solution pour résoudre ce type de labyrinthe, il est inutile d'implémenter un algorithme qui recherche le plus court chemin comme *Dijkstra*.

Architecture de la vue

La classe Vue représente la vue principale de l'application. Cette classe hérite de la classe Gtk::Window et contient plusieurs membres privés, dont des widgets GTK tels que des boutons, des étiquettes et des champs de saisie.

Elle possède également un objet Grille qui représente la grille de l'application. La classe Vue fournit des méthodes pour associer des fonctions à différents événements, tels que le redimensionnement de la grille, le clic sur le bouton "Fusion" et le clic sur le bouton "Aldous Broder".

Elle fournit également des méthodes pour récupérer la largeur et la hauteur de la grille à partir des champs de saisie correspondants. Le fichier "vue.cpp" implémente les méthodes déclarées dans le fichier "vue.hpp". Ces méthodes sont utilisées pour connecter les signaux des widgets aux fonctions correspondantes. Par exemple, la méthode *link_new_size* permet d'associer une fonction à exécuter lorsqu'une nouvelle taille est spécifiée via les champs de saisie et le bouton "Restart".

Le but principal de la Vue est de customiser l'interface graphique à nos besoin et de permettre au Contrôleur de lier les entrées utilisateur et leur logique associées.

Architecture du contrôleur

Le contrôleur est une classe qui gère la logique du jeu et agit comme un intermédiaire entre la vue (interface utilisateur) et le modèle (le labyrinthe). Son rôle principal est de recevoir les interactions de l'utilisateur à travers la vue, de traiter ces interactions et de mettre à jour le modèle en conséquence.

Voici un aperçu général du fonctionnement du contrôleur :

- **Initialisation :** Le constructeur du contrôleur crée une instance du labyrinthe (classe Labyrinth), de la vue (classe Vue) et des éléments de la fenêtre contextuelle (Gtk::Window, Gtk::Label, Gtk::Button). Il configure également la fenêtre contextuelle pour afficher le message de victoire.
- **Liaison des signaux et des événements :** Le contrôleur relie les signaux et les événements provenant de la vue aux fonctions de traitement correspondantes. Par exemple, il lie l'événement de changement de taille de la vue à une fonction qui recrée le labyrinthe avec la nouvelle taille et met à jour la vue en conséquence.
- **Traitement des interactions utilisateur :** Le contrôleur définit des fonctions de traitement pour les différentes interactions utilisateur, telles que le déplacement du joueur à l'aide des touches de direction, la fusion des cellules du labyrinthe, la construction du labyrinthe selon l'algorithme d'Aldous-Broder, etc. Lorsque ces interactions se produisent dans la vue, le contrôleur est informé et appelle la fonction de traitement appropriée.
- **Mise à jour du modèle :** Après avoir traité les interactions utilisateur, le contrôleur met à jour le modèle (le labyrinthe) en fonction de ces actions. Par exemple, lorsque le joueur se déplace, le contrôleur met à jour la position du joueur dans le modèle.
- **Mise à jour de la vue :** Une fois que le modèle a été mis à jour, le contrôleur demande à la vue de se redessiner pour refléter les changements. Par exemple, lorsque le joueur se déplace, le contrôleur met à jour la position du joueur dans la vue et demande à la vue de se redessiner.
- **Affichage de la fenêtre contextuelle :** Lorsque le joueur atteint la position finale, le contrôleur affiche la fenêtre contextuelle de victoire.
- **Gestion de la fenêtre contextuelle :** Le contrôleur gère également l'événement de clic sur le bouton dans la fenêtre contextuelle. Lorsque le bouton est cliqué, le contrôleur cache la fenêtre contextuelle.

Donc le contrôleur coordonne les actions de la vue et du modèle, en traitant les interactions utilisateur, mettant à jour le modèle et en demandant à la vue de se redessiner. Il agit comme un médiateur entre la vue et le modèle, permettant une séparation claire des responsabilités et assurant un fonctionnement cohérent du jeu.

Autre Classe

Grille

La classe Grille est une représentation graphique d'une grille utilisée pour afficher des labyrinthes ou des jeux de plateau. Elle permet d'offrir les fonctionnalités suivantes :

- **Gestion de la taille de la grille :** La classe permet de définir le nombre de lignes et de colonnes de la grille lors de sa création. Cela permet d'adapter la taille de la grille à vos besoins spécifiques.
- **Rendu graphique :** La classe utilise la bibliothèque GTK+ et la fonction *on_draw* pour effectuer le rendu graphique de la grille. Cette méthode est appelée automatiquement lorsqu'il est nécessaire de redessiner la grille, par exemple lorsqu'elle est affichée pour la première fois ou lorsqu'elle est modifiée.
- **Gestion des cases :** Chaque case de la grille est représentée par un carré de taille spécifique, définie par la variable *cote_case*. Vous pouvez ajuster cette taille en utilisant la méthode *set_case_size*. La grille est construite à partir de ces cases, ce qui permet de créer une structure en grille pour votre jeu ou votre application.
- **Gestion des éléments de la grille :** La grille peut contenir différents éléments, tels que des murs, des portes, un point de départ et une sortie. Ces éléments sont représentés graphiquement lors du rendu de la grille. Par exemple, les murs peuvent être dessinés en utilisant des lignes blanches, tandis que le point de départ et la sortie peuvent être dessinés en utilisant des formes et des couleurs spécifiques.
- **Interaction utilisateur :** La classe Grille réagit aux événements de clic de souris et de pression de touches. Vous pouvez utiliser ces événements pour interagir avec la grille, par exemple pour déplacer un personnage dans le jeu ou pour modifier les éléments de la grille en réponse aux actions de l'utilisateur.

La classe Grille fournit une interface graphique pour afficher et interagir avec une grille. Elle prend en charge le rendu graphique des cases et des éléments de la grille, ainsi que la gestion des événements utilisateur. Cela en fait un outil polyvalent pour la création de labyrinthes, de jeux de plateau et d'autres applications basées sur une structure en grille.

Observable

L'Observable est un design pattern qui permet la communication entre objets. Il est utilisé lorsque certains objets, appelés observateurs, doivent être informés des changements survenus dans un autre objet, appelé sujet.

La classe Observable est un modèle générique qui peut être utilisé avec n'importe quel type T. Elle maintient une liste d'observateurs qui sont intéressés par les changements survenus dans le sujet. La classe Observable fournit deux méthodes principales :

- **notifierObservateurs(T info):** Cette méthode est utilisée pour notifier tous les observateurs en appelant leur méthode update avec l'information fournie. L'information peut être de n'importe quel type T et est transmise aux observateurs afin qu'ils puissent prendre des mesures appropriées en réponse au changement.
- **ajouterObservateur(Observateur<T> *observateur):** Cette méthode permet d'ajouter un observateur à la liste des observateurs de l'objet Observable. L'observateur est ajouté à la fin de la liste et recevra les notifications ultérieures concernant les changements survenus dans le sujet.

Donc le design pattern Observable permet d'établir une communication de type "publier/souscrire" entre un sujet et ses observateurs. Le sujet notifie les observateurs des changements survenus, leur permettant ainsi de réagir et de prendre des mesures appropriées. Cela favorise la modularité et la réutilisabilité du code, en séparant les préoccupations entre le sujet et ses observateurs. Cette implémentation n'a pas pu être réalisée.

Observé

L'Observateur est un design pattern qui fait partie du modèle de conception Observer. Il permet à un objet, appelé observateur, d'être informé des changements survenus dans un autre objet, appelé sujet, sans que l'observateur ait à connaître les détails de mise à jour du sujet.

La classe Observateur est un modèle générique qui peut être utilisé avec n'importe quel type T. Elle définit une méthode virtuelle pure *update(const T &info)*, qui doit être implémentée par les classes dérivées. Cette méthode est appelée lorsque le sujet effectue une mise à jour et reçoit l'information mise à jour en paramètre.

Dans le labyrinthe, l'Observateur est utilisé pour permettre aux différentes parties du programme de réagir aux changements survenus dans la grille du labyrinthe. Par exemple, lorsque le joueur se déplace dans le labyrinthe, le modèle peut notifier les observateurs, tels que l'interface utilisateur, pour mettre à jour l'affichage en conséquence.

Donc l'Observateur est un design pattern qui permet aux objets observateurs d'être notifiés des changements survenus dans un sujet sans qu'ils aient à connaître les détails de mise à jour. Cela favorise la modularité, la réutilisabilité et la séparation des préoccupations dans un système logiciel. Cette implémentation n'a pas pu être réalisée.

Position du joueur

La classe Position représente la position d'un joueur dans le labyrinthe. Elle est utilisée pour stocker les coordonnées x et y du joueur dans le labyrinthe.

La classe Position a deux attributs privés : x et y, qui représentent les coordonnées x et y de la position du joueur respectivement. La classe Position fournit des méthodes pour accéder aux coordonnées x et y, ainsi que pour déplacer le joueur dans différentes directions : gauche, droite, haut et bas. Les méthodes *deplacerGauche(), deplacerDroite(), deplacerHaut()* et *deplacerBas()* ajustent les coordonnées x et y de la position en fonction de la direction de déplacement.

La position du joueur est importante dans le labyrinthe car elle permet de connaître sa position actuelle et de mettre à jour cette position lorsqu'il se déplace à l'intérieur du labyrinthe. La position est utilisée pour vérifier les collisions avec les murs, les portes et d'autres objets du labyrinthe, ainsi que pour afficher le joueur à la bonne position dans l'interface utilisateur.

Donc la classe Position représente la position du joueur dans le labyrinthe et fournit des méthodes pour accéder aux coordonnées x et y, ainsi que pour déplacer le joueur dans différentes directions. Elle est essentielle pour suivre et gérer la position du joueur dans le labyrinthe.