

Sepanta Farzollahi

## Algorithmique avancée

### Projet labyrinthe

#### Introduction

Le projet « **Labyrinthe** » vise à modéliser et résoudre un problème historique lié au royaume d'**Ayutthaya** en **Thaïlande**. Le scénario implique la punition des sujets qui ne se convertissaient pas au bouddhisme **Theravada** sous le règne du roi **Ramathibodi Ier**. Les prisonniers devaient échapper à une pièce enflammée pour être pardonnés. Le projet utilise des représentations graphiques pour les plans de pièces et propose des algorithmes pour déterminer si les prisonniers avaient une chance de survie.

#### Choix du langage : C++



Le choix du langage de programmation **C++** pour la résolution du problème du labyrinthe a été guidé par plusieurs considérations spécifiques à ce langage.

- 1) **Performance critique** : Le problème du labyrinthe implique des calculs intensifs liés à la propagation du feu et aux déplacements du prisonnier. **C++** offre un contrôle fin sur la gestion de la mémoire et une performance élevée, ce qui est crucial pour la résolution efficace du problème.
- 2) **Manipulation efficace des structures de données** : **C++** offre des fonctionnalités avancées pour la manipulation des structures de données, ce qui est essentiel pour représenter efficacement le labyrinthe sous forme de graphe pondéré et pour la mise en œuvre d'algorithmes de recherche de chemins.
- 3) **Bibliothèques standard** : Les bibliothèques standard de **C++** ont été utilisées pour des opérations courantes, offrant des fonctionnalités prêtes à l'emploi et facilitant le développement.
- 4) **Inspiration du TP7-8 en Java** : Le code fourni par **M. Lobry** pour les classes « **WeightedGraph** » et « **App** » du **TP7-8** était en **Java**, un langage orienté objet. En choisissant **C++**, qui partage également une approche orientée objet, il était possible de s'inspirer de ce code tout en adaptant et en étendant les fonctionnalités pour répondre aux exigences spécifiques du problème du labyrinthe. Cette transition facilitée entre **Java** et **C++** a permis de capitaliser sur les concepts objets tout en profitant des spécificités de **C++** pour une résolution efficace du problème.
- 5) **Inspiration du code en C** : Le code source du fichier **labyrinthe.c** disponible sur le site a également été une source d'inspiration pour le projet. En examinant cette implémentation en **C**, des idées et des approches ont été extraites, contribuant ainsi à la conception et à la réalisation du projet en **C++**. Ceci a enrichi la compréhension du problème et a permis d'explorer différentes façons de résoudre les défis spécifiques posés par le labyrinthe.

## Méthodologie

- **Compréhension du problème** : Avant de commencer le développement, une phase de compréhension approfondie du problème a été réalisée. Cela comprenait l'analyse du scénario historique du royaume d'**Ayutthaya**, la description de la punition des prisonniers et les contraintes associées au déplacement du feu et des prisonniers dans le labyrinthe.
- **Modélisation du problème** : La modélisation du problème a été réalisée à l'aide de graphes pondérés, où les sommets représentent les positions possibles dans le labyrinthe et les arêtes représentent les déplacements autorisés. La classe « **WeightedGraph** » a été créée pour encapsuler ces concepts, avec des classes internes « **Edge** » et « **Vertex** » représentant les arêtes et les sommets respectivement.
- **Choix des algorithmes** : Deux algorithmes classiques de recherche de chemin ont été sélectionnés pour résoudre le problème : **Dijkstra** et **A\***. **Dijkstra** a été choisi pour sa simplicité et sa capacité à trouver le chemin le plus court, tandis qu'**A\*** a été retenu pour son efficacité dans la recherche de chemins optimaux en tenant compte d'une heuristique.
- **Implémentation des algorithmes** : Les algorithmes **Dijkstra** et **A\*** ont été implémentés dans la classe « **Labyrinth** », héritant de la classe « **WeightedGraph** ». Chacune des méthodes a été soigneusement commentée pour expliquer son fonctionnement et sa logique. La gestion des différents cas, tels que la propagation du feu, les mouvements des prisonniers et les règles spécifiques du problème, a été intégrée dans les algorithmes.
- **Gestion des erreurs** : Une attention particulière a été portée à la gestion des erreurs. Le code vérifie la validité de l'entrée utilisateur, détecte les erreurs telles que la présence multiple de caractères spécifiques dans le labyrinthe, et fournit des messages d'erreur informatifs pour guider l'utilisateur. Ainsi, des boucles de saisie sécurisées ont été mises en place pour garantir que les données entrées, telles que le nombre d'instances et les dimensions du labyrinthe, soient valides.
- **Commentaires** : Tout au long du processus de développement, des commentaires ont été ajoutés pour expliquer la logique, les décisions de conception et le fonctionnement des différentes parties du code.

## Implémentation

L'implémentation de ce projet a nécessité une approche méthodique et la prise en compte de plusieurs considérations. Une des premières étapes a été la conception de la structure de données pour représenter le graphe pondéré et les opérations associées. La classe « **WeightedGraph** » a été développée pour encapsuler les fonctionnalités relatives aux sommets, aux arêtes, et au graphe lui-même. La division en sous-classes comme « **Edge** » et « **Vertex** » a permis une organisation claire du code.

Pour garantir la flexibilité et la réutilisabilité, les structures de données standard du langage, telles que les vecteurs et les listes, ont été largement exploitées. L'utilisation de listes chaînées pour représenter les listes d'adjacence des sommets a permis des opérations efficaces d'ajout et de suppression d'arêtes.

Une attention particulière a été portée à la gestion des données temporelles, notamment les temps individuels des sommets, les temps de parcours depuis la source, et les heuristiques. L'intégration de la bibliothèque `<limits>` a été cruciale pour définir des valeurs d'infini, facilitant ainsi les comparaisons et les mises à jour appropriées des temps.

Le processus d'implémentation a également impliqué des tests approfondis pour s'assurer que les algorithmes fonctionnaient correctement dans divers scénarios. Ainsi des jeux de données de test ont été réalisés pour évaluer la robustesse du programme, en mettant l'accent sur les cas limites et les conditions aux limites du problème.

## Fichiers sources

**WeightedGraph.h** : Il définit la classe « **WeightedGraph** », qui représente un graphe pondéré. Il contient également les classes internes « **Edge** » et « **Vertex** » utilisées pour modéliser les arêtes et les sommets du graphe.

Il contient ainsi des méthodes telles que : « **addVertex** » pour ajouter un sommet au graphe avec des attributs spécifiés et « **addEdge** » pour ajouter une arête pondérée entre deux sommets du graphe.

La classe interne « **Edge** » permet de représenter une arête pondérée avec une source, une destination, et un poids. La classe interne « **Vertex** » quant à elle permet de représenter un sommet avec des informations telles que le temps individuel, le temps depuis la source, la fonction heuristique, le sommet précédent, la liste d'adjacence, et les coordonnées.

Des conteneurs de la **STL** (**std::vector** et **std::list**) sont utilisés pour stocker les sommets et les arêtes du graphe, permettant une gestion dynamique et efficace de la structure des données.

**WeightedGraph.cpp** : Les méthodes déclarées dans « **WeightedGraph.h** » sont implémentées dans cette classe. Cela inclut l'ajout de sommets, l'ajout d'arêtes, et la réinitialisation du graphe.

Les méthodes de « **WeightedGraph** » sont utilisées pour construire le graphe pondéré du labyrinthe, en ajoutant des sommets et des arêtes en fonction des caractéristiques du problème. Ainsi la gestion des sommets et des arêtes, y compris leur ajout et leur suppression, est réalisée de manière efficace grâce aux fonctionnalités de la **STL**.

Les détails d'implémentation sont encapsulés dans cette classe, assurant une bonne modularité et facilitant l'extension du code pour d'autres problèmes.

**Labyrinth.h** : Il définit la classe « **Labyrinth** », qui hérite de la classe « **WeightedGraph** ». Cette classe contient des méthodes spécifiques pour résoudre le problème du labyrinthe, telles que **Dijkstra**, **AStar**, **distance**, etc.

Il contient ainsi des méthodes telles que : « **Dijkstra** » pour implémenter l'algorithme de **Dijkstra** afin de trouver le chemin le plus court entre deux points dans le labyrinthe, « **AStar** » pour implémenter l'algorithme **A\*** pour la recherche de chemin en tenant compte d'une heuristique, « **distance** » pour calculer la distance euclidienne entre deux points dans le labyrinthe, « **canMove** » pour vérifier si le déplacement vers un sommet est possible, « **MovementDirectionForEachTurn** » pour déterminer la direction de mouvement pour chaque tour, « **movePrisoner** » pour effectuer le déplacement du prisonnier en fonction de la direction donnée, « **winMove** » pour vérifier si le mouvement actuel conduit à la victoire, « **firePropagationAround** » pour propager le feu autour d'un sommet spécifique et enfin « **runInstance** » pour exécuter une instance du problème du labyrinthe.

**std::set** est utilisé pour gérer l'ensemble des sommets à visiter dans les algorithmes. **std::vector** et **std::list** sont utilisés pour stocker les résultats et les listes d'adjacence des sommets.

La classe « **Labyrinth** » étend la classe de base « **WeightedGraph** », montrant une utilisation efficace de l'héritage pour réutiliser les fonctionnalités existantes tout en introduisant des fonctionnalités spécifiques au problème du labyrinthe.

**Labyrinth.cpp** : Les méthodes spécifiques au problème du labyrinthe déclarées dans « **Labyrinth.h** » sont implémentées dans cette classe.

Les méthodes de la classe de base « **WeightedGraph** » sont utilisées pour construire et manipuler le graphe pondéré associé au labyrinthe. Ainsi les algorithmes de **Dijkstra** et **A\*** sont implémentés pour résoudre le problème du labyrinthe en trouvant le chemin optimal entre le point de départ et la sortie.

Les méthodes liées aux mouvements du prisonnier, à la propagation du feu et à la vérification des conditions de victoire sont implémentées de manière à résoudre le problème spécifique.

**std::cout** est utilisé pour afficher les résultats des différentes instances du problème du labyrinthe.

**main.cpp** : Il est le fichier principal qui contient la fonction **main**. Il est responsable de l'interaction utilisateur, de la gestion des instances du problème du labyrinthe, et de l'affichage des résultats.

**std::vector** est utilisé pour stocker les résultats de chaque instance du problème du labyrinthe. Tandis que **std::istringstream** est utilisé pour extraire des entiers à partir d'une chaîne de caractères lors de la saisie utilisateur.

L'utilisateur est invité à entrer le nombre d'instances du problème du labyrinthe à résoudre. Une boucle de saisie valide est mise en place pour garantir une entrée correcte sans espaces.

Les dimensions du labyrinthe et les configurations pour chaque instance sont saisies par l'utilisateur avec des vérifications d'entrée.

Pour chaque instance, un labyrinthe est créé, configuré, et résolu en utilisant la classe « **Labyrinth** ». Ainsi, les résultats de chaque instance sont stockés dans un vecteur. Une fois toutes les instances résolues, les résultats sont affichés à l'utilisateur.

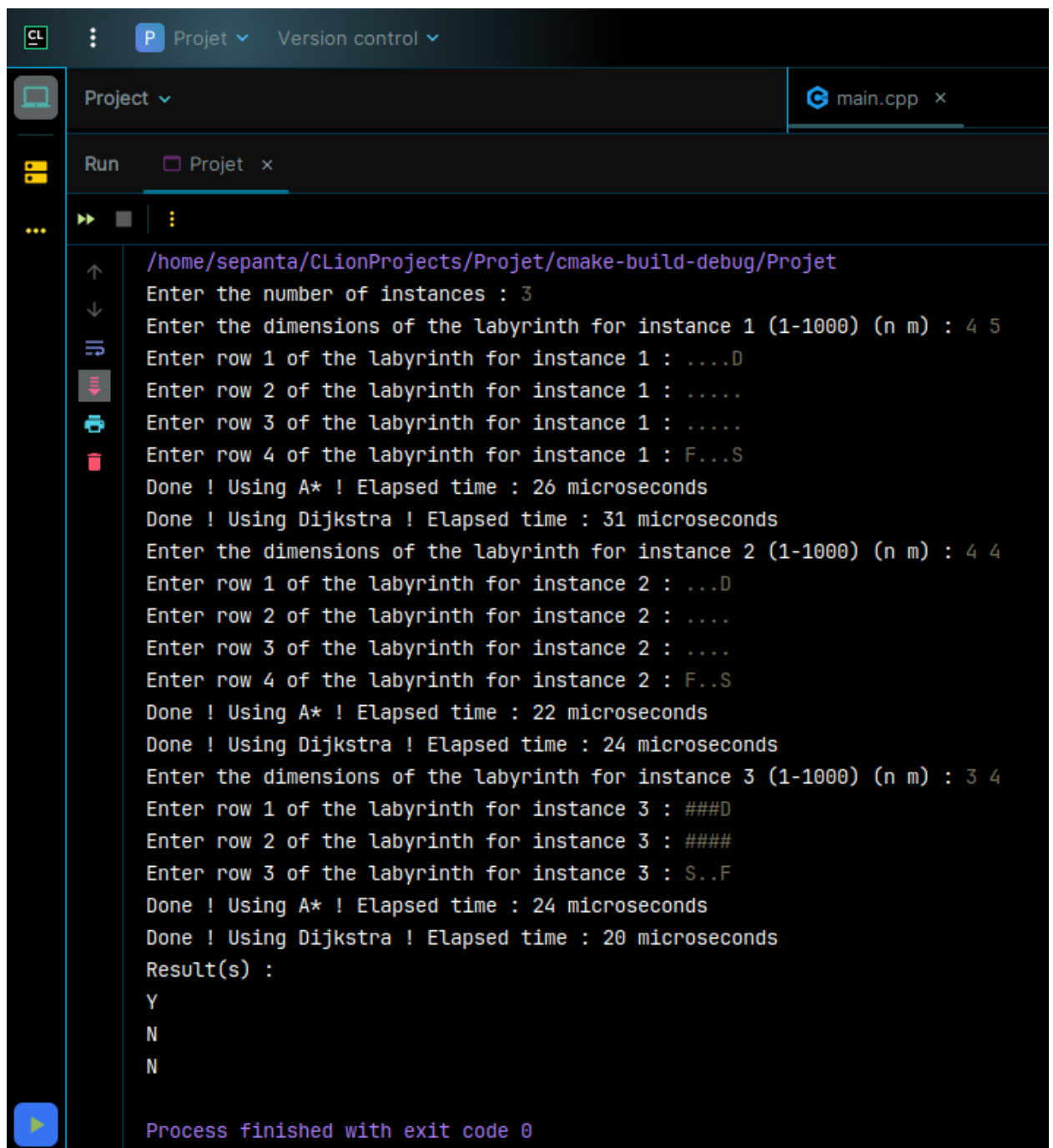
La mémoire allouée dynamiquement pour les labyrinthes est libérée après chaque instance pour éviter les fuites de mémoire.

Les fonctionnalités de la classe « **Labyrinth** » sont utilisées pour résoudre le problème du labyrinthe de manière organisée et réutilisable.

En cas d'erreurs lors de la saisie utilisateur ou de conditions invalides, les messages d'erreur sont affichés via **std::cerr**.

### Exemple d'exécution

Dans la page suivante, vous trouverez une illustration détaillée du déroulement du programme en utilisant l'exemple fourni dans le sujet (cf. image [result.png](#)) :



```
CLion
P Projet Version control
main.cpp x
Run
/home/sepanta/CLionProjects/Projet/cmake-build-debug/Projet
Enter the number of instances : 3
Enter the dimensions of the labyrinth for instance 1 (1-1000) (n m) : 4 5
Enter row 1 of the labyrinth for instance 1 : ....D
Enter row 2 of the labyrinth for instance 1 : .....
Enter row 3 of the labyrinth for instance 1 : .....
Enter row 4 of the labyrinth for instance 1 : F...S
Done ! Using A* ! Elapsed time : 26 microseconds
Done ! Using Dijkstra ! Elapsed time : 31 microseconds
Enter the dimensions of the labyrinth for instance 2 (1-1000) (n m) : 4 4
Enter row 1 of the labyrinth for instance 2 : ...D
Enter row 2 of the labyrinth for instance 2 : ....
Enter row 3 of the labyrinth for instance 2 : ....
Enter row 4 of the labyrinth for instance 2 : F..S
Done ! Using A* ! Elapsed time : 22 microseconds
Done ! Using Dijkstra ! Elapsed time : 24 microseconds
Enter the dimensions of the labyrinth for instance 3 (1-1000) (n m) : 3 4
Enter row 1 of the labyrinth for instance 3 : ####
Enter row 2 of the labyrinth for instance 3 : ####
Enter row 3 of the labyrinth for instance 3 : S..F
Done ! Using A* ! Elapsed time : 24 microseconds
Done ! Using Dijkstra ! Elapsed time : 20 microseconds
Result(s) :
Y
N
N
Process finished with exit code 0
```

## Conclusion

Ce projet de résolution de labyrinthe a été une exploration captivante des algorithmes de recherche de chemin (**Dijkstra** et **A\***) dans un contexte historique. En utilisant le langage **C++**, le développement du code a été facilité par la familiarité avec les concepts de la programmation orientée objet. L'utilisation du paradigme objet a permis une modélisation claire et efficace du graphe pondéré, contribuant à une implémentation structurée et réutilisable.

Les choix de conception, tels que la représentation du labyrinthe en tant que graphe pondéré, ont été motivés par la nécessité de résoudre le défi posé par **Márcio "l'indispensable" Himura**. Les algorithmes **Dijkstra** et **A\*** ont été soigneusement adaptés pour résoudre le problème spécifique de la fuite à travers un labyrinthe enflammé.

L'inspiration tirée du code en langage **Java** de **M. Lobry** pour le **TP7-8** a facilité la compréhension des concepts clés, tandis que l'étude du code en langage **C** a fourni des perspectives supplémentaires pour aborder le problème. Ces influences externes ont été judicieusement intégrées, montrant la valeur de la collaboration et de l'apprentissage continu.

En résumé, ce projet m'a permis d'appliquer des connaissances en programmation et algorithmique à un scénario historique intrigant. Les défis rencontrés ont été autant d'opportunités d'apprentissage, et le résultat final illustre l'efficacité de l'approche choisie. Ce travail contribue à renforcer les compétences en résolution de problèmes informatiques tout en explorant des problèmes du monde réel à travers le prisme de la programmation.

### Remarques

**N.B.** Pour ce projet, **CLion** a été choisi comme **IDE** pour la programmation en **C++**. Ce choix a été motivé par la compatibilité de **CLion** avec **CMake**, un système de gestion de projet très utile. La version **C++23** du langage **C++** a été privilégié, et la version **3.27** de **CMake** a été utilisé pour garantir la compatibilité avec mon environnement de développement.

**N.B.** Pour compiler et exécuter les fichiers **C++** qui ont été générés avec **CMake** (le répertoire **Projet\_cmake**), vous devrez d'abord, vous assurez d'avoir **CMake** installé sur votre système. Si ce n'est pas déjà le cas, vous pouvez télécharger **CMake** depuis le site officiel : <https://cmake.org/download/>. Une fois **CMake** installé, vous pouvez compiler les fichiers source **C++** en utilisant les commandes « **cmake .** » qui générera les fichiers nécessaires pour la compilation et « **make** » qui effectuera la compilation proprement dite. Une fois la compilation terminée avec succès, vous pouvez exécuter le fichier exécutable en utilisant la commande « **./Projet** ».

**N.B.** Le programme peut être aussi compilé en utilisant le compilateur **C++ g++** (le répertoire **Projet\_g++**) avec les options **-std=c++11**, **-Wall** et **-Wextra** pour assurer la conformité aux normes et détecter les avertissements potentiels : « **g++ -std=c++11 -Wall -Wextra -o projet \*.cpp -lm** ». L'option **-lm** indique au compilateur de lier avec la bibliothèque mathématique (**libm**). La commande de compilation générera un exécutable nommé « **projet** » à partir de tous les fichiers source présents dans ce répertoire courant. Pour exécuter le programme, utilisez la commande « **./projet** ». Assurez-vous que l'exécutable « **projet** » a été correctement généré après la compilation. L'exécution du programme lancera son exécutable « **projet** », permettant ainsi de tester les fonctionnalités implémentées.