POLYTECH SORBONNE

Sorbonne Université
2022-2023

# Petit concours de programmation en C

## Labyrinthe

Le jeu *Labyrinthe* est un jeu de société créé par Max Jünger Kobbert en 1986, et édité par Ravensburger[1]. Il a été édité dans de nombreuses versions (de Star Wars à la reine des neige, en passant par des variantes 3D, duel, etc. ).

Le but du jeu est simple : dans un labyrinthe sans cesse en mouvement, il vous faut aller récupérer tous les trésors avant vos adversaires. Le labyrinthe est composé de tuiles indiquant les murs et les passages. À chaque tour de jeu, avant de se déplacer, le joueur peut insérer une tuile de jeu pour faire se déplacer une rangée (ligne ou colonne) du labyrinthe, pour se rapprocher de son prochain trésor, et/ou empêcher l'adversaire d'accéder au sien.
Nous allons jouer avec une version simplifié du jeu, avec seulement 2 joueurs et des trésors connus de tous.


Le but du projet est d'écrire un programme capable de jouer au jeu Labyrinthe, en respectant les règles. Ce programme pourra jouer contre d'autres programmes (du plus stupide au plus malin), et pourra participer au *Petit tournoi de programmation* qui aura lieu mi-janvier.

[1] `https://tinyurl.com/Laby-Ravensburger`

# 1 Règles du jeu

Le plateau de jeu est composé de L×H tuiles qui forment le labyrinthe[2]. Elles sont placées en début de partie aléatoirement sur le plateau.
Les tuiles sont des morceaux de labyrinthe qui indiquent où sont situés les murs et les passages. Il y a globalement trois types de tuiles: en L, en T, et en I.



Ces tuiles peuvent aussi être tournées quatre fois (d'un quart de tour), et peuvent aussi être porteuses d'un trésor. Dans le jeu, le trésor est donné par un nombre, entre 1 et 24 (il y a donc 24 trésors à trouver).

Chaque joueur est représenté par un pion sur ce labyrinthe. Le but du jeu est alors d'aller récupérer succcessivement avec son pion tous les trésors (dans l'ordre croissant de 1 à 24 pour le premier joueur, dans l'ordre décroissant pour le second[3]).

En plus des L×H tuiles du labyrinthe, il y a une tuile supplémentaire, mise de côté, qui est utilisé pour modifier le labyrinthe en cours du jeu.

Chaque tour de jeu se déroule de la façon suivante :

- Modification du labyrinthe : Dans les colonnes et les lignes impaires (les indices commençant à zéro), il est possible d'insérer la tuile supplémentaire afin de faire coulisser toutes les tuiles d'une rangée ou colonne et ainsi modifier le labyrinthe. La tuile ainsi expulsée devient la nouvelle tuile supplémentaire qui sera utilisée au tour suivant.
  **Attention :** Une tuile ne peut jamais être réintroduite à l'endroit même où elle vient d'être expulsée (et donc d'annuler le coup précédent).
  Si en faisant coulisser une rangée ou colonne, un joueur (son pion) est expulsé du plateau, alors son pion est placé à l'opposée sur la tuile qui vient d'être insérée (cesi ne compte pas comme un déplacement de pion).

- Déplacement du pion : après avoir modifié le labyrinthe, le joueur peut déplacer son pion jusqu'à n'importe quelle case en suivant un couloir ininterrompu (en suivant le labyrinthe). Il peut se déplacer aussi loin qu'il veut, et peut aussi ne pas se déplacer.
  Si un joueur atteint le trésor recherché, il l'a trouvé, et il pourra passer au trésor suivant (ordre croissant ou décroissant) pour son prochain tour de jeu.

La partie s'arrête lors qu'un joueur a trouvé les 24 trésors du plateau.

# 2 Principe général

Il y a ici un serveur qui s'occupe de gérer les différentes parties (et s'occupera du tournoi). Chacun devra écrire un programme qui, en utilisant les fonctions

[2]Dans le jeu de plateau, L=H=7. Ici, on aura $7 \leq \mathtt{L} \leq 13$ et $7 \leq \mathtt{H} \leq 13$.

[3]Dans le jeu de société, chaque joueur a des cartes lui indiquant quel est le prochain trésor à récupérer, mais ce trésor n'est pas connu de l'adversaire. Ici, il faut aller récupérer successivement tous les trésors, et l'adversaire sait quel est le prochain trésor, et peut donc jouer de façon à ne pas avantager son adversaire.

proposées, pourra communiquer avec ce serveur pour

- démarrer une partie
- récupérer la taille du labyrinthe, les différentes tuiles qui le composent, etc.
- jouer un coup
- récupérer le coup de l'adversaire

Votre programme devra donc démarrer une partie et succcessivement jouer un coup et récupérer le coup de l'adversaire jusqu'à la fin de la partie. Votre programme devra donc connaitre à chaque instant l'état du labyrinthe (et donc mettre à jour le labyrinthe au fur et à mesure des coups joués).

# 3 Description de l'API

Tout d'abord, un serveur permettant de gérer les parties (avec la règle du jeu) a été mis en place. Ce serveur (*Coding Game Server*), qui est utilisé tous les ans pour un projet de ce genre, gère les différentes parties avec le calcul des scores et le respect des règles du jeu, l'affichage au format texte de la partie, les différents *bots* qui vous permettront de tester vos programmes, et même l'organisation de tournoi (ainsi qu'un serveur web pour afficher tout un tas d'informations).

Ensuite, une petite librairie vous est donnée, avec les fichiers suivants et ce document[4]:

- `LabyrinthAPI.h` qui contient les types et prototypes des fonctions que vous devez utiliser pour jouer au jeu ;
- `LabyritnhAPI.c` qui contient son implémentation (qui se base sur `clientAPI.c`, et `clientAPI.h`). Vous n'avez pas spécialement à regarder le contenu de ses fichiers (et encore moins à les modifier).

L'API proposée est susceptible d'évoluer au cours du temps (pas les prototypes, juste les implémentations), ainsi que le serveur, pour rajouter de nouvelles fonctionnalités (l'affichage peut d'évoluer) et bien sûr corriger tous les *bugs* qui pourraient arriver (je vous indiquerai s'il y a besoin de faire un `git pull`).

N'hésitez pas à me faire remonter tout bug ou problème par email (`thibault.hilaire@lip6.fr`) ou sur Discord, en précisant le problème, comment il arrive et en précisant (si possible) le numéro de la partie (les parties sont logguées).

**N'oubliez pas de lire les commentaires du fichier `LabyrinthAPI.h`, car il contient tous les détails d'utilisation des fonctions.**

## 3.1 Les types

Quelques types structurés vous sont fournis pour vous aider à communiquer avec le serveur et à jouer. Veuillez vous référer au fichier pour les détails (les commentaires et les noms utilisés sont en anglais[5]; vous pouvez suivre cette convention ou non pour votre programme).

[4] Repository Git : `https://github.com/thilaire/Labyrinthe-Polytech.git`

[5] Des fautes d'orthographe doivent trainer; n'hésitez pas à me les signaler pour que je les corrige!

- `t_return_code` : indique si un coup joué est normal (`NORMAL_MOVE`), donc le jeu continue, ou bien si le coup provoque la fin du jeu et qu'on a gagné (`WINNING_MOVE`) ou perdu (`LOSING_MOVE`). Ce type est défini dans `clientAPI.h`.
- `t_move` : définit un coup. Un coup est défini par
  - `insert` : l'endroit où sera insérer la tuile. Ce champ est de type `t_insertion` et peut prendre les valeurs `INSERT_LINE_LEFT`, `INSERT_LINE_RIGHT`, `INSERT_COLUMN_TOP` et `INSERT_COLUMN_BOTTOM` pour indiquer que la tuile est insérée sur une ligne par la gauche, une ligne par la droite, une colonne par le haut ou une colonne par le bas, respectivement.
  - `number` qui indique le numéro de la ligne ou de la colonne où on insère la tuile (`number` doit impérativement être impair, sinon le coup n'est pas valide).
  - `rotation` qui indique de combien de quart de tour (entre 0 et 3) dans le sens des aiguilles d'une montre il faut faire avant d'insérer la tuile supplémentaire;
  - `x` et `y` qui indique les coordonnées d'où se déplace le pion.

  Un `t_move` contient aussi six champs qui sont mis à jour une fois que le coup a été joué :
  - `tileN`, `tileE`, `tileS`, `tileW`, `tileItem` qui donne la tuile qui a été expulsée par le coup (`tileN`, `tileE`, `tileS` et `tileW` indique s'il y a un mur au Nord, Est, Sud et Ouest respectivement; `tileItem` indique s'il y a un trésor et sa valeur: il vaut 0 s'il n'y a pas de trésor sur cette tuile, et sinon vaut l'indice du trésor entre 1 et 24).
  - `nextItem` indique le numéro du prochain trésor pour le joueur qui vient de jouer son coup.

Il n'y a pas de type "tuile" de créer, car ce sera probablement à vous d'en créer un, comme bon vous semble. Dans l'API, chaque tuile est donc représentée par 5 entiers : 4 pour la présence des murs au nord, est, sud et ouest, ainsi qu'un entier pour l'indice du trésor (ou 0 s'il n'y a pas de trésor).

## 3.2 Fonctions

Vous avez accès aux fonctions suivantes (dont les prototypes se trouvent dans `LabyrinthAPI.h`, ainsi que les détails dans les commentaires) :

- `connectToServer` et `closeConnection` pour se connecter au serveur et fermer la connection; ce sera la première et dernière chose à faire dans le programe.
- `waitForLabyrinth` permet d'attendre que le serveur nous place dans une partie, et donne les 1ères informations nécessaires (pour préparer la récupération des données de jeu).
  - Le champ `gameType` définit la partie que l'on veut faire (partie contre un `bot`, dans un tournoi ou se placer en attente de partie; on n'utilisera pas cette dernière fonctionnalité pour le moment). Les détails d'utilisation sont donnés dans la section 4.1 ;

  - `labyrinthName` permet de récupérer le nom de la partie ;
  - `sizeX` et `sizeY` permet de récupérer la taille du labyrinthe.

- la fonction `getLabyrinth` permet de récupérer les données du labyrinthe (dans un tableau de `sizeX*sizeY*5` entiers, chaque tuile étant représentée par 5 entiers : la présence d'un mur au nord, à l'est, au sud et à l'ouest, ainsi que le numéro du trésor de la tuile. On récupère aussi les informations de la dernière tuile). Il est clair que cette façon de stocker le labyrinthe `n'est pas` la bonne façon de faire : l'idée ici est de vous obliger à trouver une bonne façon de représenter le labyrinthe et les tuiles.

- la fonction `sendMove` qui permet de jouer un coup (en donnant un `t_move`), alors que la fonction `getMove` permet de récupérer le coup de l'adversaire.

- `printLabyrinth` permet d'afficher le plateau de jeu (l'affichage se fait en mode texte, avec quelques couleurs. La position des pions est donné par un carré à fond rouge ou bleu. Les trésors par des drapeaux, avec une couleur bleue ou rouge pour le prochain trésor de chaque joueur).

- `sendComment` permet d'envoyer, au cours du jeu, des commentaires pour l'adversaire (parfaitement inutile, mais à utiliser en tournoi pour narguer son adversaire !).

# 4 Premiers programmes

On procède par étapes. On donne ici les indications pour créer pas à pas un programme capable de jouer à Labyrinthe.

## 4.1 Se connecter

Tout d'abord, on cherche à se connecter au serveur (avec la fonction `connectToServer`) et à créer une partie (avec la fonction `waitForLabyrinth`) qui nous donne en retour le nom de la partie créée et la taille du labyrinthe.
On peut se connecter au serveur pour une partie libre avec `gameType` une chaîne de caractères vide, ou bien s'entraîner contre un *bot* avec `"TRAINING <BOT>"` où l'on remplace `<BOT>` par le nom d'un *bot*. Sont disponibles actuellement deux *bots*:

- le bot `DONTMOVE` qui insert aléatoirement la tuile et ne bouge pas;

- le bot `RANDOM` qui insert aléatoirement la tuile et se déplace vers le prochain trésor s'il peut (sinon ne bouge pas).

Au moins un autre bot devrait arriver prochainement (bot qui joue un coup lui permettant de se déplacer jusqu'au prochain trésor).

Il est aussi possible de passer des options à la création de partie en complétant la chaine `gameType` avec des paramètres sous la forme `"key=value"`, séparés par un espace. Les options suivantes sont possibles:

- `timeout` : permet de définir le temps maximum (en secondes) pour jouer avant d'être déclaré perdant (par défaut 10s). À modifier (passer à 1000) quand vous voulez jouer manuellement ;

- `seed` : permet de définir la graine du générateur de nombres aléatoire. En utilisant toujours la même graine, on obtient toujours la même partie (ce qui est très utile pour débugger). De plus, le nom de la partie contient la graine à utiliser (les $6^{ers}$ caractères définissent, en hexadécimal, la graine; cela permet de rejouer une partie à l'identique) ;
- `start` : permet de définir si on commence (`"start=0"`) ou non ;
- `margin` : permet de rajouter un espace entre les lignes et les colonnes dans l'affichage du labyrinthe s'il vaut 1 (vaut 0 par défaut);
- `display` : affiche en plus les données du labyrinthe sous forme de chiffre si sa valeur vaut `"debug"`. Dans ce cas, chaque tuile est affiché par 5 nombres (accolés) représentant les mur nord, est, sud, ouest ainsi que l'indice du trésor. Cela peut être très utile pour vérifier que le programme que vous écrivez a la bonne représentation du labyrinthe : pour cela, il vous faut écrire votre propre fonction d'affichage (en suivant le format proposé) et de comparer les valeurs. Cela est bien plus rapide que de comparer votre affichage du labyrinthe avec celui proposé par le serveur.

Il y aura deux serveurs qui tourneront à l'adresse

`172.105.76.204`

et sur les ports 1234 et 5678. Si l'un est à l'arrêt, l'autre devrait répondre.

Une fois connecté, on peut récupérer les données du labyrinthe avec la fonction `getLabyrinth`, à qui il faut donner un tableau de $N$ entiers, où $N$ est égal à 5 fois le nombre de tuiles (données par `sizeX` et `sizeY`). On s'occupera de ces données plus tard

La fonction `getLabyrinth` renvoie aussi notre numéro de joueur (0 ou 1), sachant que le joueur n°0 est celui qui commencera à jouer.

On n'oubliera pas de se déconnecter du serveur (fonction `closeConnection`) en fin de programme.

**Question 1** *Écrivez un programme qui démarre une partie avec le* bot *`DONTMOVE`, récupère les données, et se déconnecte.*

## 4.2 Boucle de jeu

On peut maintenant avoir une boucle de jeu permettant de jouer, à tour de rôle, avec un *bot*.
Il faut donc une boucle où l'on va :

- afficher labyrinthe et l'état du jeu (utile pour débugger) avec la fonction `printLabyrinth` ;
- si c'est à notre tour de jouer, on va jouer un coup (fonction `PlayMove`).
- si c'est au tour de l'adversaire, on récupère son coup, avec la fonction `getMove` qui remplit un `t_move` (dont on ne s'occupera pas dans cette section).

Cette boucle de jeu ne s'arrête que lorsqu'un coup joué par un des deux joueurs n'a pas renvoyé la constante `NORMAL_MOVE` (mais `WINNING_MOVE` ou `LOSING_MOVE` comme défini dans le type `t_return_code`).

**Question 2** *Complétez le programme précédent en permettant de jouer une partie, jusqu'à sa fin (un joueur perd). Écrivez une fonction qui demande à l'utilisateur de choisir un coup (et toutes les données nécessaires pour jouer un coup). Il n'y a pas besoin de vérifier les données entrées (ce sera vous l'utilisateur). Le coup à jouer sera demandé à l'utilisateur par quelques* scanf.
*On ne s'occupera pas pour le moment du coup de l'adversaire ou des valeurs remplies par l'appel à* `PlayMove`.
*Testez et jouez une partie contre le* bot.

## 4.3 Structure de données et découpage du code

On voit bien dans notre programme que beaucoup de données concernant le jeu ont été créées (par encore toutes remplies ou utilisées) : les tuiles du labyrinthe ainsi que la tuile supplémentaire, la position des joueurs, l'indice du prochain trésor qu'ils doivent trouver, etc.
Nous allons évidemment créer des fonctions pour gérer les différents coups possibles (mettre à jour le labyrinthe, la position des joueurs, etc.), et il nous faut donc créer les structures de données pour faciliter l'échange entre les fonctions et organiser tout cela.

**Question 3** *Créer la ou les structures de données pour stocker les données de la partie, le labyrinthe, une tuile, etc.*

Le but des questions suivantes est de gérer les différents coups possibles (que l'on joue ou que l'adversaire joue) et de mettre à jour les données concernant la partie (labyrinthe, etc.).

L'idée est d'avoir les structures de données nécessaires pour connaître l'état du jeu et être capable de calculer les prochains coups à jouer.

## 4.4 Initiation des données

En début de partie, grâce aux fonctions `WaitForLabyrinth` et `getLabyrinth`, on récupère les informations nécessaires pour initialiser la partie (les joueurs sont dans les coins supérieur gauche et inférieur droit, respectivement).

**Question 4** *Écrire la fonction qui initialise le jeu avec les données de départ.*

## 4.5 Mettre à jour les informations du jeu à chaque coup

Puisque nous avons deux joueurs qui vont jouer des coups définis par une variable de type `t_move`, nous devons maintenant écrire une fonction qui met à jour les données du jeu (tuiles du labyrinthe, position des joueurs, indice du trésor, etc. ) en fonction d'un coup joué. Pour comparer votre labyrinthe avec celui du serveur, le mode `display=debug` donné à la fonction `waitForLabyrinth` pourra être utile.

# 5 Écrire un programme qui joue seul

Maintenant que vous avez toute la structure de données, et qu'elle est à jour à chaque coup, vous allez pouvoir remplacer votre fonction qui demande à l'utilisateur

de rentrer au clavier un coup, par une fonction qui va trouver un coup à jouer pour remplir les objectifs.

Une façon de faire (mais pas la seule, pas forcément la meilleure) est de tester tous les modifications de labyrinthe possibles, et pour chacune d'elle de vérifier si on peut se déplacer jusqu'au trésor prochain. Pour cela, il vous faudra une fonction capable de calculer tous les points que l'on peut atteindre depuis une position de départ, et donc de vérifier si le pion peut atteindre le prochain trésor. Vous pouvez vous inspirer de ce qui a été fait dans le TP labyrinthe...

L'étape suivante serait aussi de vérifier que les modifications de labyrinthe ne permettent pas facilement à l'adversaire d'atteindre son trésor...

À vous de ~~jouer~~ programmer !!!

Et n'hésitez pas à aller sur les salons Discord associés pour poser vos questions et avoir de l'aide si nécessaire !