Mini-Projet : Labyrinthe

L'objectif du projet est de programmer un jeu de Labyrinthe de façon à ce que votre programme puisse jouer une partie contre les programmes des autres étudiants.

1 Organisation générale du projet

Ce projet se déroulera en binôme (ou trinôme si nombre impair). Le choix des binômes est libre. La soutenance du projet aura lieu en deux temps :

- une soutenance orale de 15 minutes (avec supports Power-Point (ou Beamer/LaTeX) le **17 Janvier 2016** ;
- un tournoi de Labyrinthe entre les binômes. Le gagnant du tournoi aura l'honneur de rencontrer un programme réalisé par un enseignant et invaincu (par définition) jusqu'à aujourd'hui. Les modalités détaillées de la soutenance seront précisées mi-décembre. Le tournoi sera commun entre les spécialités ROB et MAIN.

1.1 Déroulement du projet

Le projet se déroulera à partir du 29 Novembre. Vous bénéficierez de 3 séances de TP encadrées pour la réalisation de ce projet. L'encadrant de TP aura le rôle de chef de projet. C'est donc lui qui vous conseillera et vous guidera pour les étapes importantes, et c'est aussi lui qui validera la réalisation de chacune des étapes du projet.

1.2 Assistance

À tout moment il vous est possible de poser une question sur la plateforme Piazza dédiée à l'adresse suivante :

`https://piazza.com/upmc.fr/fall2016/laby/home`

Vous allez très rapidement recevoir un email pour vous y inscrire.
Cette plateforme permet à chacun de poser des questions sur un forum dédié au projet. Cela vous permettra de poser vos questions en dehors des séances de TP encadrées et d'obtenir des conseils ou de l'aide. Vous pouvez aussi utiliser ce forum entre vous pour vous aider (vous aider et non pas sous-traiter votre code !). N'hésitez pas à l'utiliser !

1.3 Évaluation

Le projet sera évalué sur les critères suivants :

- L'avancement du projet au fur et à mesure (validation des étapes) ;
- La soutenance orale ;
- La qualité du programme final (bugs, fonctionnalités codées, etc.) ;
- La place au tournoi de Labyrinthe.

2 Règles du jeu

2.1 Le but du jeu

Vous êtes deux adversaires dans un labyrinthe. Un trésor (et un seul) est présent au milieu du labyrinthe.

L'objectif est d'être le premier à récupérer le trésor.

2.2 Données d'une partie

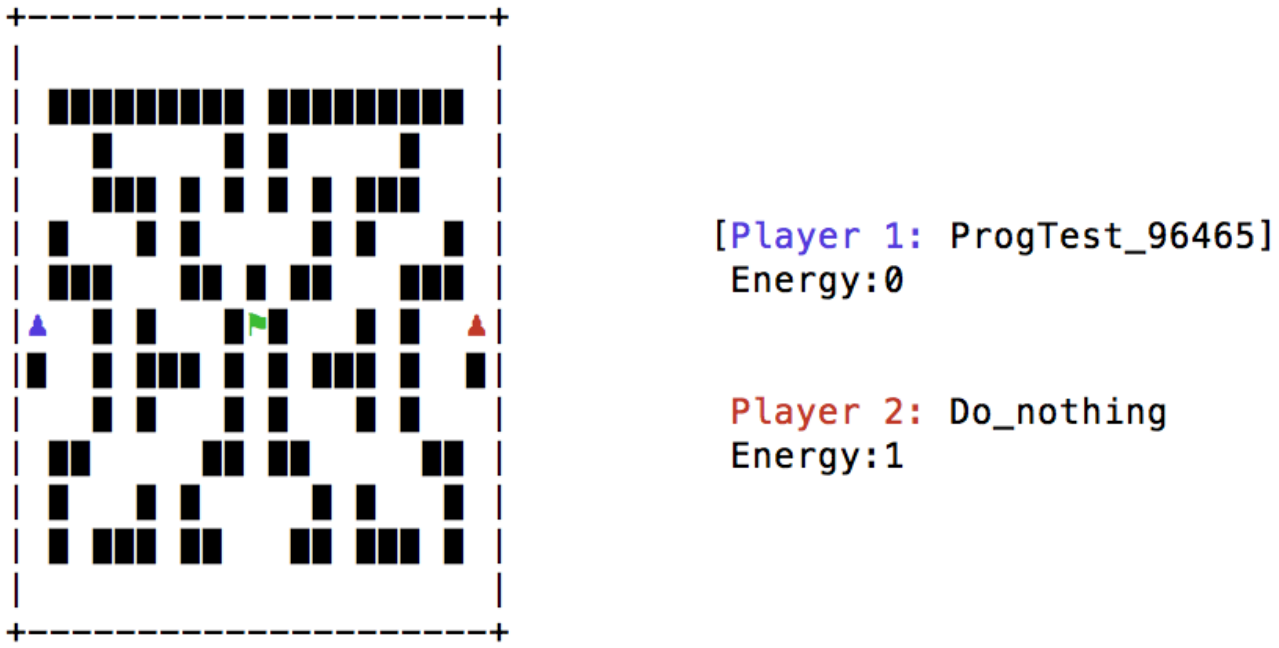

FIG. 1 – Exemple d'un labyrinthe en début de partie. La case de coordonnées (0,0) est située en haut à gauche.

En début de partie le serveur de jeu vous envoie les informations suivantes :

- La taille du labyrinthe en hauteur H et longueur L via la fonction `waitForLabyrinth` (cf 4.2). La taille est aléatoire et telle que L et H soient impairs, $13 \leq L \leq 17$ et $9 \leq H \leq 15$;
- Le nom du labyrinthe ;
- Les données du labyrinthe : Les positions des murs (infranchissables) et des espaces libres. Sur la figure 1, les murs sont indiqués par des carrés noirs. Pour plus de détail, voir la fonction `getLabyrinth` (cf 4.2) ;
- L'indication du joueur qui commence.

Ensuite, vous pouvez construire le labyrinthe initial sachant que :

- Le joueur qui commence est situé sur la case de coordonnées $x = 0$ et $y = H/2$ (division entière) ;
- L'autre joueur est situé sur la case de coordonnées $x = L - 1$ et $y = H/2$ (division entière) ;
- Le trésor est situé sur la case de coordonnées $x = L/2$ et $y = H/2$ (division entière) ;
- Le joueur qui commence a un niveau initial d'énergie de 0, l'autre joueur a un niveau initial d'énergie de 1.

Notez que les coordonnées $x = 0$ et $y = 0$ correspondent à la case située en haut à gauche.

Ensuite, à chaque tour de jeu, vous pouvez soit effectuer un déplacement classique de votre joueur, soit (sous certaines conditions) effectuer une rotation du labyrinthe.

2.3 Déplacement classique d'un joueur

À chaque tour vous pouvez vous déplacer d'une case vers une case libre (c'est-à-dire sans mur) vers le haut, le bas, la gauche ou la droite. Vous pouvez aussi rester sur place. La figure 2 montre un exemple de déplacement.

- À chaque fois que vous faites un déplacement de ce type, vous gagnez une unité d'énergie ;
- Vous ne pouvez pas vous déplacer sur un mur, par contre, vous pouvez vous déplacer sur votre adversaire, ou sur le trésor (dans ce cas vous le récupérez) ;
- Le labyrinthe est circulaire (et même toroïde !), un joueur sortant à gauche se retrouve tout à droite et un joueur sortant en haut se retrouve en bas.

2.4 Rotation du labyrinthe

À chaque tour vous pouvez dépenser 5 unités d'énergie pour décaler une ligne (ou une colonne) entière du labyrinthe d'une case vers la gauche ou vers la droite (haut/bas pour la colonne). La figure 3 présente un exemple de rotation.

- La rotation est circulaire : la case sortante d'un côté entre de l'autre ;
- Les joueurs ou le trésor se déplacent avec la rotation du labyrinthe.

2.5 Conditions de fin de partie

Conditions de victoire :

- Vous atteignez le trésor en premier ;
- Votre adversaire perd avant.

Conditions de défaite :

- Votre adversaire atteint le trésor avant vous ;
- Vous faites un coup illégal (ex : déplacement dans un mur) ;
- Vous tentez de faire une rotation, mais n'avez pas assez d'énergie ;
- Vous ne respectez pas le séquencement du programme (cf section 3.2) ;
- Vous mettez trop de temps à jouer (max. 10 secondes).

3 Structure du programme

Votre programme Labyrinthe devra se connecter à un serveur de jeu qui a deux rôles :

1. Mettre en relation des programmes afin de les faire jouer entre eux ;
2. Vérifier la validité des coups joués et déclarer le joueur gagnant.

3.1 Fichiers fournis

Tous les fichiers qui vous sont fournis dans le cadre de ce projet sont disponible sur le serveur de Polytech aux emplacements `/home/sasl/encad/brajard/projet/CGS_distrib` et `/home/sasl/encad/brajard/projet/CGS_lib`. Il vous est conseillé de :

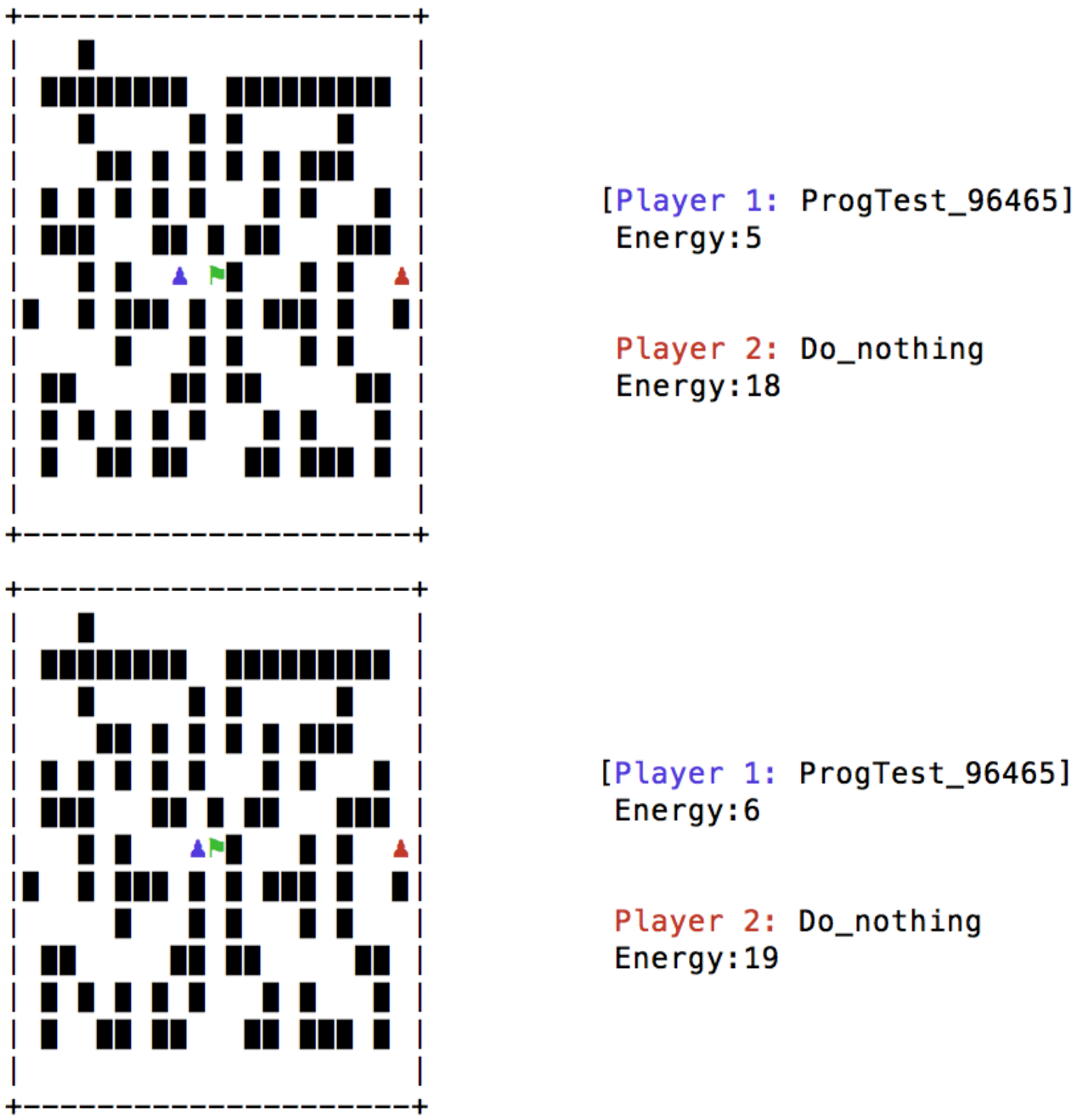

Fig. 2 – Déplacement du joueur 1 vers la droite

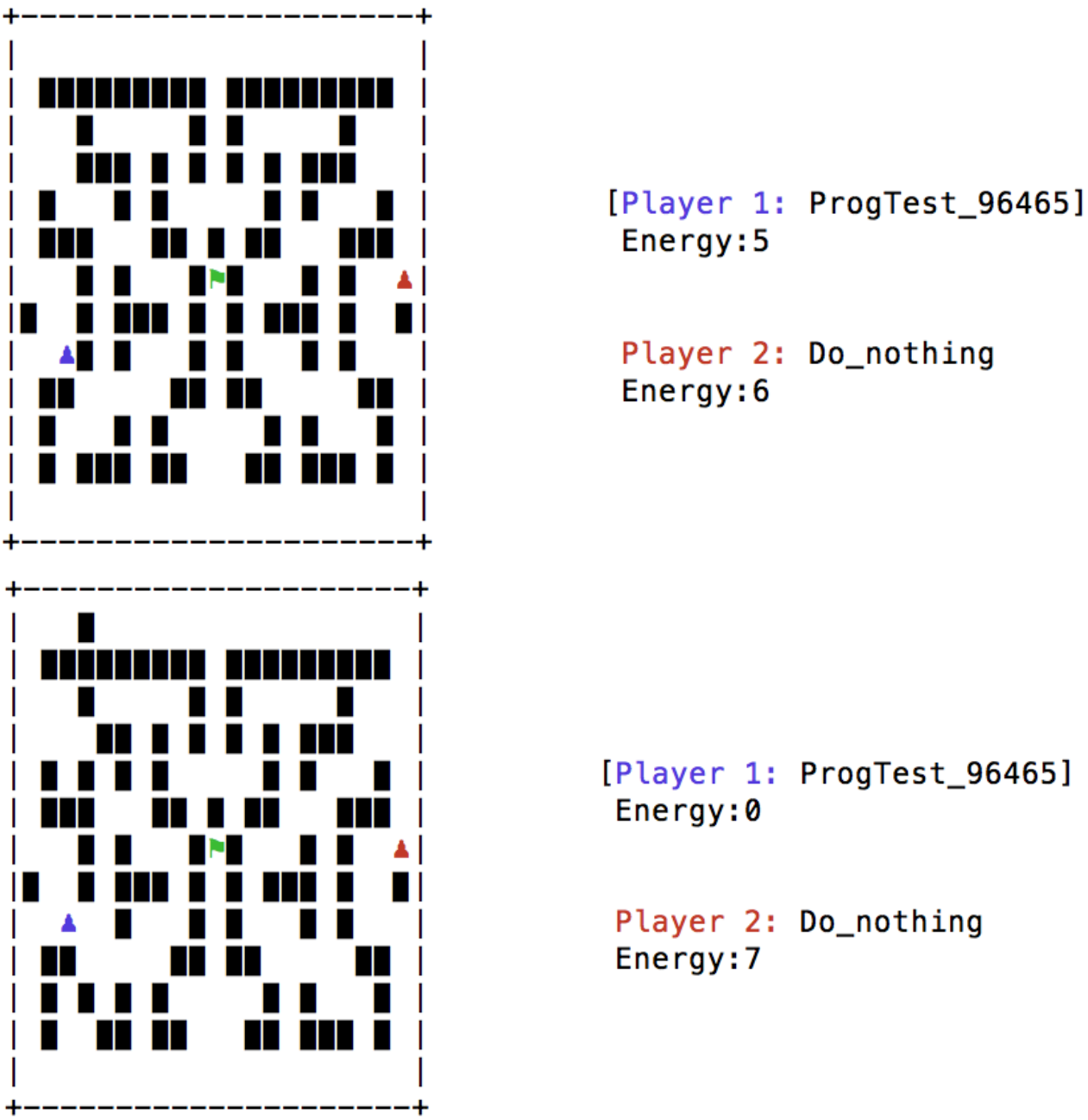

FIG. 3 – Rotation de la colonne 3 vers le haut

- Copier sur votre compte les fichiers `template.c` et `makefile` fournis que vous pourrez modifier (tout le contenu du répertoire `CGS_distrib`)
  Par exemple, si vous êtes dans le répertoire dans lequel vous souhaitez réaliser votre projet, il suffit de faire :
  `cp /home/sasl/encad/brajard/projet/CGS_distrib/* .` (n'oubliez pas le `.` à la fin) ;
- Ne pas changer les chemins vers les *header* et *bibliothèques* indiquées dans le `makefile` (ne pas copier le contenu du sous-répertoire `CGS_lib` car il pourra potentiellement être mis à jour si nécessaire). Attention les bibliothèques ont été compilées sur les machine de polytech et ne fonctionnent donc qu'à Polytech (ou sur une machine équivalente).

On donne ci-après la structure générale de votre programme. Les fonctions appelées et les types utilisés sont détaillés dans la section 4. Les modèles de structure et les prototypes des fonctions sont décrits dans le fichier `CGS_lib/include/labyrinthAPI.h`
Voici les différentes étapes que devra respecter votre programme pour interagir avec le serveur.

3.2 Étapes à respecter dans votre programme

Ces étapes sont en grande partie (mais pas complètement) codées dans le fichier `template.c` qui vous est fourni en début de projet (voir section 3.1)

1. Se connecter au serveur en appelant la fonction `connectToServer` ;
2. Demander un Labyrinthe en appelant la fonction `waitForLabyrinth` :
   - Pour jouer contre un joueur par défaut toujours disponible, le spécifier avec le paramètre "training"
     Ex : `training="DO_NOTHING_GAME timeout=n"` (en spécifiant une valeur pour `n` (cf 3.3). D'autres adversaires par défaut peuvent être spécifiés.
   - Pour joueur contre un autre programme, vous devez laisser l'option `training` vide (`""`). Vous devez alors vous rendre sur la page web du serveur via un navigateur (ex : `http://pc4021.polytech.upmc.fr:8088` pour un serveur tournant sur le pc4021) et créer une rencontre entre vous et un autre joueur. Attention la page web n'est accessible que depuis une machine du réseau Poytech'.
3. Récupérer les données du labyrinthe en appelant la fonction `getLabyrinth` ;
4. Répéter les actions suivantes :
   - Si c'est à vous de jouer : envoyer votre coup en appelant la fonction `sendMove`, vous récupérez le code `ret`.
   - Si c'est à l'adversaire de jouer : récupérer le coup joué par l'adversaire en appelant la fonction `getMove`, vous récupérez également le code `ret`.
   - En fonction de `ret` :
     - `ret==MOVE_OK` : vous pouvez continuer à jouer ;
     - `ret==MOVE_LOSE` : le joueur qui vient de jouer à perdu ;
     - `ret==MOVE_WIN` : le joueur qui vient de jouer à gagné.
5. Fermer la connexion au serveur en appelant la fonction `closeConnection`.

De plus, à tout moment, vous pouvez demander l'affichage du jeu à l'écran en appelant la fonction `printLabyrinth`.

3.3 Les joueurs par défaut

Lors de l'appel de la fonction `waitForLabyrinth`, il est possible de demander un joueur par défaut ("training player"). Vous pourrez alors tester votre programme sans qu'un autre joueur

ne se connecte sur le serveur. Pour cela, il faut spécifier dans une chaîne de caractère `training` le nom de ce joueur par défaut, ainsi que les paramètres. Pour l'instant, deux joueurs par défaut sont disponibles :

- **DO_NOTHING** : Ce joueur ne fait aucun mouvement.
  Exemple : `training = "DO_NOTHING timeout=10"` pour le joueur DO_NOTHING avec une limite de temps pour jouer (voir conditions de défaites 2.5) de 10 sec (Ce temps peut être changé).
- **PLAY_RANDOM** : Ce joueur joue aléatoirement, mais tous ses coups sont valides.
  Exemple : `training = "PLAY_RANDOM timeout=60 rotation=True"` pour jouer avec 60 s de limite de temps et la possibilité pour le joueur PLAY_RANDOM de faire les rotations de lignes et colonnes dans le labyrinthe (mettre l'option à `False` si on veut que PLAY_RANDOM ne fasse que des déplacements classiques.)
- **...** : D'autres joueurs par défaut viendront s'ajouter en cours de projet.

4 API

4.1 Les structures à utiliser

La structure `t_move`

Cette structure décrit un coup joué par vous ou par votre adversaire, elle est utilisée dans les fonctions `sendMove` et `getMove` :

```
/*
A move is a tuple (type,value):
- type can be ROTATE_LINE_LEFT, ROTATE_LINE_RIGHT,
   ROTATE_COLUMN_UP,
ROTATE_COLUMN_DOWN, MOVE_LEFT, MOVE_RIGHT, MOVE_UP or MOVE_DOWN
- in case of rotation, the value indicates the number of the line
(or column) to be rotated
*/
typedef struct
{
  t_typeMove type; /* type of the move */
  int value; /* value associated with the type
          (number of the line or the column to rotate)*/
} t_move;
```

Par exemple :

- Une variable `m` décrivant le déplacement d'une case vers la droite a pour valeurs `m.type = MOVE_RIGHT` et `m.value = -1` (valeur non utilisée).
- Une variable `m` décrivant une rotation de la colonne 3 vers le haut a pour valeurs `m.type = ROTATE_COLUMN_UP` et `m.value = 3`. On rappelle ici que la première colonne (et la première ligne) est numérotée 0.

La structure `t_return_code`

Cette structure décrit le code retour des fonctions `sendMove` et `getMove`. Elle permet de décrire si un coup qui vient d'être joué est correct (`MOVE_OK`), gagnant (`MOVE_WIN`) ou perdant

(`MOVE_LOSE`). Cette valeur doit être testée dans votre programme pour savoir si vous devez rejouer ou si la partie est terminée.

4.2 Fonctions fournies

`connectToServer`

Permet de se connecter au serveur (à faire une seule fois, en début de programme).

```
/* ------------------------------------
 * Initialize connection with the server
 * Quit the program if the connection to the server
 * cannot be established
 *
 * Parameters:
 * - serverName: (string) address of the server
 *   (it could be "localhost" if the server is run in local,
 *   or "pc4521.polytech.upmc.fr" if the server runs there)
 * - port: (int) port number used for the connection
 * - name: (string) name of the bot : max 20 characters
 *         (checked by the server)
 */
void connectToServer( char* serverName, int port, char* name);
```

Le numéro de port et le nom du serveur vous seront donnés lors de la première séance.

`closeConnection`

Permet de se déconnecter du serveur (à la fin du programme).

```
/* ----------------------------------
 * Close the connection to the server
 * to do, because we are polite
 *
 * Parameters:
 * None
*/
void closeConnection();
```

`waitForLabyrinth`

Permet d'attendre une partie Labyrinth, et de récupérer son nom et sa taille. Permet aussi de spécifier si on veut jouer contre un joueur d'entraînement.

```
/*
   ------------------------------------------------------------

 * Wait for a Game, and retrieve its name and first data
 * (typically, array sizes)
 *
 * Parameters:
```

```
 * - training: string (max 50 characters) type of the training
 *             player we want to play with
 *             (empty string for regular game)
 * - labyrinthName: string (max 50 characters),
 *                  corresponds to the game name
 * - sizeX, sizeY: sizes of the labyrinth
 *
 * training is a string like "NAME key1=value1 key2=value1 ..."
 * - NAME can be empty. It gives the type of the training player
 * - key=value pairs are used for options
 *   (each training player has its own options)
 *   invalid keys are ignored, invalid values leads to error
 *   the following options are common to every training player
 *   (when NAME is not empty):
 *        - timeout: allows an define the timeout
 *                   when training (in seconds)
 * the NAME could be:
 * - "DO_NOTHING" to play against DO_NOTHING player
 *   (player that does not move)
 * - "PLAY_RANDOM" for a player that make random (legal) moves
 *   (option "rotation=False/True")
 */
void waitForLabyrinth( char* training, char* labyrinthName,
           int* sizeX, int* sizeY);
```

Les arguments `labyrinthName`, `sizeX` et `sizeY` sont passés via des pointeurs et leurs valeurs seront donc données en sortie de fonction.

`getLabyrinth`

Permet de récupérer les données initiales du labyrinthe.

```
/* ------------------------------------
 * Get the labyrinth and tell who starts
 * It fills the char* lab with the data of the labyrinth
 * 1 if there's a wall, 0 for nothing
 *
 * Parameters:
 * - lab: the array of labyrinth
 *   (the pointer data MUST HAVE allocated with the right size
    !!)
 *
 * Returns 0 if you begin, or 1 if the opponent begins
 */
int getLabyrinth( char* lab);
```

L'argument `lab` est passé via un pointeur et la valeur sera donc donnée en sortie de fonction. Le tableau `lab` rempli par cette fonction est un tableau à 1 dimension de taille $L \times H$ qui contient des entiers (sur 8 bits). Les cases valent 0 pour les cases libres et 1 pour les murs. Il est rempli dans le sens des lignes, c'est-à-dire que les L premières valeurs correspondent aux L cases de la

ligne 0, les L suivantes correspondent à la ligne 1, etc. Les L dernières valeurs du tableau `lab` correspondent ainsi à la ligne $H - 1$.

`getMove`

Permet de récupérer le coup joué par l'adversaire.

```
/* ----------------------
 * Get the opponent move
 *
 * Parameters:
 * - move: a move
 *
 * Returns a return_code
 * MOVE_OK for normal move,
 * MOVE_WIN for a winning move, -1
 * MOVE_LOSE for a losing (or illegal) move
 * this code is relative to the opponent (MOVE_WIN if HE wins,
    ...)
 */
t_return_code getMove( t_move* move );
```

L'argument `move` est passé via un pointeur et la valeur sera donc donnée en sortie de fonction.

`sendMove`

Permet de jouer un coup.

```
/* -----------
 * Send a move
 *
 * Parameters:
 * - move: a move
 *
 * Returns a return_code
 * MOVE_OK for normal move,
 * MOVE_WIN for a winning move, -1
 * MOVE_LOSE for a losing (or illegal) move
 * this code is relative to your programm (MOVE_WIN if YOU win,
    ...)
 */
t_return_code sendMove( t_move move );
```

L'argument `move` est passé en entrée de la fonction.

`printLabyrinth`

Permet d'afficher le labyrinthe.

```
/* ----------------------
 * Display the labyrinth
 * in a pretty way (ask the server what to print)
```

```
 */
void printLabyrinth();
```

5 Les étapes

Votre projet sera suivi par votre responsable de TP qui s'assurera que vous complèterez les différentes étapes d'achèvement de votre projet. Les étapes (dans l'ordre) sont les suivantes :

1. Programme qui ne fait rien (équivalent du "do nothing"). Ce programme consiste en une petite modification du fichier `template.c` fourni de façon à intégrer la boucle du jeu et les conditions de fin de partie. Le programme doit jouer le coup vide (`DO_NOTHING`).
2. Programme où vous indiquez "à la main" le coup que vous voulez jouer. Ce programme (avec un timeout adapté pour vous laisser le temps) vous permettra d'indiquer le coup joué à la main (à l'aide par exemple d'un `scanf`). Le but étant de se familiariser avec le jeu. Inutile de "sécuriser" votre appel à la fonction `scanf`.
3. Programme qui joue de façon aléatoire. C'est une étape importante, car il faudra mettre en place les structures qui permettront à votre programme de connaître l'ensemble des coups possibles (prise en compte des murs et des rotations). Notez qu'à cette étape, votre programme sera capable de participer au tournoi prévu fin janvier (même si ses chances de victoire seront minces).
4. Programme qui joue de façon intelligente avec l'algorithme A*. Cette étape consiste à programmer un algorithme A* de façon à ce que votre programme aille vers le trésor le plus rapidement possible, mais sans effectuer aucune rotation.
5. (si possible) Programme qui joue de façon très intelligente. Ce programme devra intégrer la notion de chemin le plus court déterminé à l'étape précédente mais pourra également effectuer des rotations de labyrinthe. C'est dans cette étape-là que vous déterminez une stratégie pour votre programme.
   Pour prendre en considération les coups possibles de l'adversaire, des algorithmes de type min-max pourront être mis en œuvre.