Le train mexicain

1 Déroulement du projet

Le *train mexicain* est un jeu à plusieurs joueurs possédant de multiples variantes. Le projet consiste à implémenter différentes versions des règles ainsi que différents joueurs, des joueurs humains sur la ligne de commandes, des joueurs humains jouant sur une interface graphique, des joueurs jouant à travers le réseau et des joueurs automatiques.

2 Canevas général

Le *train mexicain* se joue avec des dominos. C'est à dire des petits pavés possédant une face séparée par deux parties sur chacune desquelles est indiqué un chiffre. L'envers des dominos est vide.

Chaque joueur du *train mexicain* possède des dominos qu'il est le seul à voir. Nous l'appellerons la *main* du joueur. En début de partie chaque joueur reçoit le même nombre de domino. 12 dominos pour 2 à 6 personnes, 10 dominos pour 7 et 8 personne, 8 dominos pour 10 personnes. Chaque joueur possède également un train de dominos. Un train est constitué de dominos mis en ligne et tel que les dominos se touchent seulement par des cotés identiques $((12, 12)(12, 11)(11, 4)(4, 3)(3, 3)(3, 2))$. En plus des trains des joueurs, un train spécial appelé le *train mexicain* est également présent. Les dominos qui ne sont dans aucun train ni dans aucune des main des joueurs sont dans la pioche, à l'envers.

Chaque joueur joue une fois par tour de jeu. Lors de son tour un joueur peut piocher, passer son tour ou poser un domino sur son propre train ou sur un train qui est ouvert. Les différentes variantes du jeu indiquent dans quel cas un joueur peut piocher, passer ou poser une pièce et dans quel cas un train est ouvert ou non.

Un jeu est constitué de plusieurs séries. Une série commence avec un double au centre de la table, c'est le premier domino de tous les trains. Une série se termine si un joueur n'a plus de pièce, ou si il n'y a plus de dominos dans la pioche et que tous les joueurs ont passé leur tour à la suite. On compte alors les scores en faisant la somme des faces des dominos que les joueurs ont encore dans leur main. Celui qui a le plus petit score a gagné.

3 Implémentation : Première partie

Cette partie correspond au rendu partiel.

Une librairie `Engine` est fournie sous forme bytecode `engine.cma` à l'adresse `www.lri.fr/~bobot/PF`.

3.1 Dominos

Cette librairie définit l'interface des dominos

```
module type Domino = sig
  type t
  (** Deux dominos qui possèdent les mêmes deux
      parties sont considérés égaux *)

  val compare : t -> t -> int
  val equal : t -> t -> bool
  val hash  : t -> int


  val d_side : t -> int * int
  (** [d_side dom] renvoit les deux parties du domino *)
```

```
  val d_match : t -> int -> int option
  (** [d_match dom side] renvoit None si aucune des parties du domino
      n'est [side] sinon renvoit [Some oside] avec [oside] l'autre
      coté du Domino *)
  val make : int -> int -> t
  (** [make side1 side2] crée le domino possédant les deux parties
      [side1] [side2] *)

end
```

La fonction `compare` définit la comparaison sur les dominos, `equal` définit l'égalité, et `hash` définit une fonction de hachage sur domino.

Question 1 Définir un module `Domino` qui corresponde à cette signature. On prendra `type t = int * int`. On s'assurera que la première composante est toujours plus petite que la seconde.

3.2 Règle

Cette librairie demande que les règles soient des modules avec l'interface `Rule` :

```
(** Règles du jeu *)
module type Rule = sig

  (** {2 Types} *)

  type t (** instances *)
  module Domino : Domino

  (** {2 Exceptions} *)

  exception BadSeat of int
  (** [BadSeat n] n n'est pas un joueur correct (n <= 0 <
      nb_joueur) *)
  exception BreakRules of string
  (** [BreakRules] ce n'est pas un déplacement autorisé *)
  exception Win of int * int list
  (** [Win (seat,scores)] le joueur [seat] win et les scores sont [scores] *)
  exception Deuce of int list
      (** [Deuce scores] les joueurs sont ex aequos avec les scores [scores] *)

  (** {2 Jeux courant} *)

  val create_instance : ?seed:int -> first:int -> int -> t
  (** [create_instance ~first n] crée une instance pour n joueurs
      commencant par le domino double [first] *)

  val hand_of : t -> seat -> Domino.t list
  (** [hand_of inst seat] retourne la main du joueur [seat] *)

  val train_of : t -> train -> int list
  (** [train_of inst train] retourne le train [train] dans
```

```
    l'ordre inverse. *)

  val is_open : t -> train -> bool
  (** [is_open t seat] indique si le train [train] est ouvert *)

  val turn_of : t -> int
  (** [turn_of inst] renvoie le numero du joueur qui doit jouer *)

  val score_of : t -> seat -> int
  (** [score_of inst seat] renvoit le score actuelle du joueur [seat] *)

  type doable = | PASS | DRAW | POSE

  val doable_of : t -> seat -> doable list
  (** [action inst seat] renvoit ce que le joueur [seat] peut faire *)

  val mattock_size : t -> int
  (** [mattoc_size inst] renvoit la taille de la pioche *)

  val nb_seat : t -> int
  (** [nb_seat inst] renvoit le nb de joueur autour de la table *)

  (** {2 Déplacement} *)

  val draw : t -> seat -> Domino.t * t
  (** [draw inst seat] le joueur [seat] demande de piocher un domino *)

  val pass : t -> seat -> t
  (** [pass inst seat] le joueur [seat] demande de passer son tour *)

  val pose : t -> seat -> train -> Domino.t -> t
  (** [pose inst seat train domino] le joueur
    [seat] demande de poser le domino [domino] sur le train [train].*)

end
```

On va définir la première version des règles. Dans cette version à chaque tour de jeu un joueur peut :

- passer son tour, c'est alors au joueur suivant de jouer
- prendre un domino de la pioche si la pioche n'est pas vide, et continuer à jouer.
- poser un domino à la suite de son train ou du train mexicain, c'est alors aux joueurs suivant de jouer

Dans cette version seul le train mexicain est ouvert.

Question 2 Implémenter cette règle simple en respectant l'interface donnée. On utilisera des structures de données persistantes (Map, Set, ...)

3.3 Joueur humain

Une fois que l'on a une règle on peut grâce au foncteur `Make_Room` fourni par la librairie `engine` créer une *pièce* dans laquelle chaque table de jeu respecte la même règle. Une *pièce* permet également de faire des joueurs spécifique à cette règle à partir de joueur généraliste.

```
type args
```

```
module Make_Room (R : Rule) = sig

  type player
  module type Make_Player (P : Player) = sig
    val player : args -> player
  end

  val play_game : player list -> int list
  (** [play_game players] renvoit le score des joueurs après qu'ils
      aient joués *)

end
```

Un joueur généraliste (module de signature `Player`) est un foncteur :

```
module type Table = ...
module type Player (T : Table) =
sig
  type t (** L'environnement du joueur *)

  val new_game : T.t -> T.token -> args -> t
  (** [new_game gt token args] commence une nouvelle partie ou le joueur aura
      le token [token] *)

  val other_draw : T.t -> t -> seat -> t
  (** [other_draw gt t seat] Le joueur [seat] a pioché *)

  val other_pass : T.t -> t -> seat -> t
  (** [other_pass gt t seat] Le joueur [seat] a passé *)

  val other_pose : T.t -> t -> seat -> train -> T.Domino.t -> t
  (** [other_pose gt t seat train domino] le joueur [seat] a posé sur
      le train [train] le domino [domino] *)

  val win : T.t -> t -> seat -> int list -> t
  (** [win gt t seat scores] le joueur [seat] a gagné, les scores de la
      manche sont [scores] *)

  val deuce : T.t -> t -> int list -> t

  val play : T.t -> t -> t * t T.action
  (** [play gt t] le joueur peut joueur un coup *)

end
```

Lorsqu'on implémente un joueur qui aura la signature `Player` on doit donner une définition aux fonctions `new_game`, `other_draw`, `other_pass`, `other_pose`, `win`, `deuce`, `deuce`, `play`. Dans la définition de ces fonctions on peut utiliser toutes les fonctions définies dans le module `T` de signature `Table` donné en argument.

Une table de jeu définit ce qu'un joueur peut voir de son environnement, les trains qui sont sur la table, la taille de la pioche, ce qu'il peut jouer. C'est le lien entre le joueur et les règles. Un joueur ne peut voir que sont propre jeu car

la fonction `hand_of` prend un `token` en argument. Or chaque joueurs ne reçoit au début de la partie qu'un unique `token` qui correspond uniquement à sa main.

Question 3 Implémenter un joueur qui sera guidé par un humain depuis la ligne de commande. Il devra afficher les trains présents sur la table sur le terminal et demander au joueur humain quel doit être son prochain coup.

Maintenant que l'on possèdent un module `Rule` et un module `Player` on peut utiliser le module `Make_Room` de cette manière :

```
module Room = Engine.Make_Room(Rule)
module RPlayer = Room.Make_Player(Player)
```

Question 4 Écrire un programme pour jouer à trois joueurs humains

3.4 Joueur Automatique

Le premier joueur automatique est glouton, il suit les règles suivantes par ordre de priorité (la plus prioritaire en premier) :

1. Poser un domino de sa main sur son train
2. Poser un domino de sa main sur un train ouvert
3. Piocher un domino
4. Passer son tour

Question 5 Implémenter ce joueur automatique en utilisant `doable_of` pour connaître les actions possibles

Question 6 Écrire un programme qui permette de choisir le nombre de joueurs et le type de joueurs sur la ligne de commande. Exemple pour sélectionner deux joueurs automatiques simples et deux joueurs humains :

```
./mexican -player greedy -player greedy -player curses -player curses
```

4 Implémentation : Deuxième partie

Une correction des parties 3.2 et 3.3 est fournie.

Pour le rendu final il faudra faire les sections 4.1, 4.2 et au choix la section 4.3 ou la section 4.4.

4.1 Règles *avec trains ouverts*

Nous allons maintenant décrire une version du Mexican train avec des règles plus contraignantes que celles précédemment données :

- Les dominos doubles sont particuliers : lorsqu'un joueur en pose un il peut continuer à jouer.
- Un joueur ne peut piocher que s'il ne peut pas poser de dominos et s'il n'a pas déjà pioché.
- Un joueur ne peut passer son tour qu'après avoir pioché et dans le cas où il ne peut poser un domino.
- Enfin dès qu'un joueur passe son tour son train est déclaré ouvert jusqu'à ce qu'il pose de nouveau un domino.

Question 7 Implémenter ces règles *avec trains ouverts* et vérifier que votre joueur automatique respecte bien ces règles.

4.2 Joueurs automatiques complexes

Maintenant nous allons réaliser un joueur automatique plus complexe.

Question 8 Écrire une fonction pour calculer la plus longue suite de dominos en utilisant les dominos de la main du joueur.

Les dominos restant seront joués en priorité à ceux de la plus longue suite.

Question 9 Programmer un joueur utilisant cette stratégie.

4.3 Interface graphique

Nous allons réaliser un joueur humain en utilisant la librairie graphics. L'interface devra permettre de sélectionner une pièce à jouer, ainsi que le train sur lequel la poser. L'interface devra également indiquer si le joueur choisit de piocher ou de passer son tour.

Question 10 Réaliser un joueur humain avec une telle interface graphique.

4.4 Jouer à travers le réseau

Le protocole Le protocole réseau est basé sur une connexion tcp/ip par échange de message. On vous fournit, protocol.ml protocol_ast.ml, pour parser les messages. A vous décrire les fonctions d'écriture de message (avec Printf par exemple).

Voici un exemple d'échange il y a trois joueurs :

```
server : New 0 endlist
client : Nb_seat
server : Nb_seat 5
client : End
server : 0 1 Other_pose 1 MT 5 12
client : End
server : 1 2 Play
client : Hand_of 0      #Le 0 est une valeur par défaut, forcement ça main
server : Hand 2 3 4 6 8 9 10 11 endlist
client : My_seat
server : Your_seat 1
client : Train_of 1
server : Train_of 12 endlist
client : Draw
server : Draw 10 12
client : Pose 1 10 12
server : Other_draw 2
client : Train_of 2
server : Train_of 12 endlist
client : End
server : ...
...
```

Pour décrire le protocole on définit les non-terminaux suivant :

- `number : [0-9]+`
- `train : "MT"|number`
- `domino : number number`
- `number_list : number ... number "endlist"`

- `string_list : string ... string "endlist"`
- `bool : "true", "false"`
- `arg : number | string`
- `arg_list : arg ... arg "endlist"`
- `doable : "Pass" | "Draw" | "Pose"`
- `doable_list : doable ... doable "endlist"`

On peut remarquer que les listes ne sont qu'une suite de non-terminaux se terminant par `endlist`

Dans ce protocole à tous moment soit le client soit le serveur attendent une réponse de l'autre. En premier c'est le client qui attend une requête du serveur. Le serveur peut lui envoyer des requêtes qui correspondent aux fonctions de la signature de player (new_game, other_draw,other_pose,win,deuce,play) :

```
"New" number arg_list
number number "Other_draw" number
number number "Other_pass" number
number number "Other_pose" number train domino
number number "Win" number number_list
number number "Deuce" number_list
number number "Play"
```

Le joueur possède habituellement son propre environnement. Il est dans le protocol symbolisé par un entier. Chaque environnement du client possèdent un identifiant unique que lui as assigné le serveur. Par exemple "New 5 endlist" demande au client d'exécuter la fonction `new_game` avec une liste d'argument vide. L'environnement renvoyé par la fonction sera identifié par le numéro 5. Dans les autres requêtes décrites précédemment, les deux nombres correspondent respectivement à l'identifiant de l'environnement initial et à l'identifiant de l'environnement final. Par exemple `5 6 Other_draw 3` demande au client d'exécuter la fonction `other_draw` avec l'environnement identifié par le numéro 5. L'environnement résultant sera identifié par le numéro 6.

Après cette requête le serveur attend une requête du client :

```
My_seat
Nb_seat
Train_of train
Is_open train
Mattock_size
Hand_of number
Doable
End
Draw
Pass
Pose train domino
```

Ces requêtes, sauf les 4 dernières, correspondent aux fonctions de `Table`. `Hand_of` possèdent un argument qui peut-être dans le cas d'un joueur mis arbitrairement étant donné qu'un joueur ne peut accéder qu'à sa propre main. La requête `End` correspond au retour des fonctions `new_game, other_draw,other_pose,win,deuce`. Les 3 dernières requêtes correspondent au retour de fonction de la fonction `play`. Dans tous les cas à part ceux de `End,Pass,Pose` le serveur répond ensuite par une de ses réponses selon le cas :

```
Your_seat number
Nb_seat   number
Train_of number_list
Hand domino_list
Is_open bool
Mattock_size number
Doable doable_list
Draw domino
```

Le serveur utilise pour répondre la Table qui est mis à sa disposition.

Les retours chariots, espaces et tabulations ne sont pas pris en compte. Nous vous fournissons une librairie pour lire le protocole

Le serveur Le serveur est un joueur générique comme un autre. On utilisera l'argument de type `args` pour connaître le port d'attente de connexion du client. Dans ce protocole le serveur n'attend qu'une seule connexion. Cependant à travers cette connexion un même joueur peut s'asseoir à plusieurs places d'une même table.

Un exemple d'environnement pour le serveur est :

```
type t = {
  id : int;
  token : T.token;
  cin : in_channel;
  cout : out_channel;
  last_id : int ref;
}
```

Où `id` est l'identifiant d'un environnement du joueur distant et `last_id` est le prochain identifiant disponible pour le prochain environnement à identifier.

Le client Le client sera un foncteur qui prendra en argument un joueur générique.

Question 11 Créer un module `Table` qui implémente un module ayant l'interface de `Engine.Table` mais qui utilise la connexion réseau pour obtenir les informations demandées

Question 12 Créer un foncteur qui prend en argument un module `Player`, qu'il instancie avec le module écrit à la question précédente. Écrire ensuite une fonction `wait_command` qui attend sur la socket une commande venant du serveur.

Question 13 Écrire une fonction `val connect : string -> int -> unit` telle que `connect server port` se connecte au serveur `server` sur le port `port`. Puis attend une commande avec la fonction `wait_command`

5 Observateurs

La librairie donne également la possibilité de définir des observateurs. Un observateur est comme son nom l'indique passif. Par contre un observateur peut connaître la main de tous les joueurs car pour lui un `token` n'est qu'un alias pour `seat`. Écrire un observateur peut être utile pour suivre le jeu de joueurs automatiques qui joueraient entre eux ou pour débugger.

6 Évaluation

L'évaluation est constituée :

1. par un rendu intermédiaire le lundi 22 novembre à 9h qui portera sur votre travail jusqu'à la section 3.4
2. par un rendu final ainsi qu'un rapport (3 pages) à soumettre le samedi 18 décembre à 12h
3. par la soutenance qui aura lieu le mercredi 5 janvier pour les L3 Info et le jeudi 6 janvier pour les L2 MIAGE.

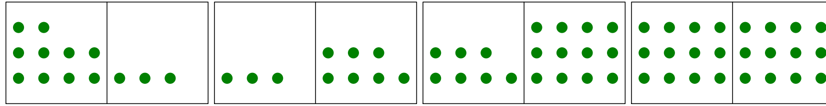
Les rendues prendrons la forme d'une archive tar qui devra être envoyée selon les modalités décrites sur la page du projet.

7 Versions

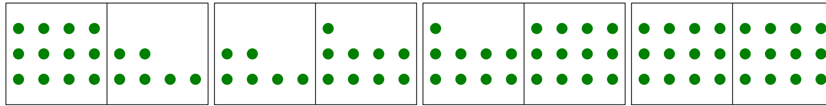
Le sujet du projet sera mis à jour si nécessaire :

- Version 1 : 13/10/2010
- Version 2 : 02/11/2010
  - Ajout des consignes pour rendre les projets
  - Précision concernant les joueurs
- Version 3 : 29/11/2010
  - Ajout de la mention de la correction
  - L'interface graphique ou le réseau deviennent facultatives.
  - Date de soutenance reculée. Le recule du rendu final en découle.

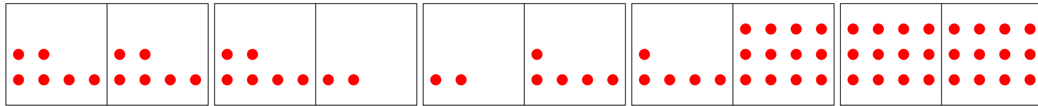
Mexican Train



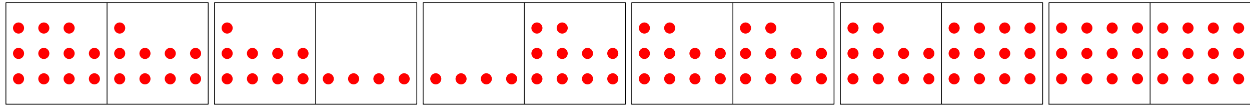
Train du joueur 0 :



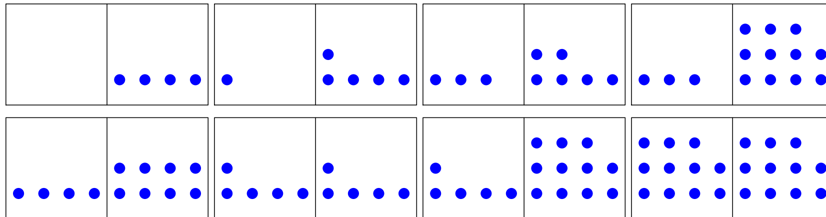
Train du joueur 1 :



Train du joueur 2 :



Main du joueur 0



Que dois-je jouer?

FIGURE 1 – Exemple d'interface graphique