BELDILMI Guillaume
GAUGUET Guillaume

Systèmes et Réseaux : Projet

Objectif

Créer un jeu du "6 qui prend" dans une version qui doit permettre, au minimum, à des joueurs humains et robots de jouer sur la même machine, où chaque carte vaut une tête de bœuf. D'autres fonctionnalités peuvent être ajoutées :

- nombre de têtes de bœufs variable par carte ;
- jeu avec utilisateurs en réseau ;
- joueurs robots plus intelligents ;
- sauvegarde des parties dans fichiers de log et production d'un document pdf montrant le déroulement de la partie ;
- génération de statistiques (par exemple points obtenus et parties gagnées/perdues par utilisateur, points moyens par parties, etc) ;
- comparaison statistique de la stratégie de joueur robot avec la stratégie de jouer les cartes au hasard.

Structure du projet

Nous avons choisi de diviser le projet en plusieurs parties :

- un programme `game` qui va gérer le déroulement de la partie ;
- une série de scripts chargés de gérer la compilation, l'exécution ainsi que la gestion du programme `game` en local ou en réseau.

Arborésence du projet

Les fichiers du projet sont organisés de la manière suivante :

```
.
├── game
│   ├── bin
│   │   └── game
│   ├── inc
│   │   └── game.h
│   ├── makefile
│   └── src
│       ├── deal_cards.c
│       ├── debug.c
│       ├── main.c
│       ├── play_bot.c
│       ├── play.c
│       ├── play_human.c
│       └── run.c
├── rapport.md
├── run
│   ├── build.sh
│   ├── client.sh
│   ├── lobby.sh
│   ├── local.sh
│   └── server.sh
└── sujet.pdf
```

Fonctionnement général

Compilation

La compilation du projet se fait via le script `build.sh`. Ce script va se charger de lancer la commande `make` dans le dossier `game`, avec la directive `re` afin de directement recompiler entièrement le projet si celui-ci a déjà été compilé.

Jeu en local

Le script `local.sh` va se charger de lancer le programme `game` en local. Le jeu sera joué par un joueur humain et plusieurs joueurs robots.

Jeu en réseau

Le fonctionnement du jeu en réseau est divisé en plusieurs parties :

- `client.sh` : ce script va se charger de se connecter au serveur pour jouer une partie, il s'utilise avec l'adresse du serveur passée en paramètre ;
- `server.sh` : ce script va se charger de lancer le serveur qui va gérer les connexions des clients ;
- `lobby.sh` : ce script va se charger d'accueillir les clients à leur connexion et de les assister dans la création d'une partie ou la connexion à une partie existante.

La communication entre les multiples clients et le programme `game` se fait via des tubes nommés. Le script `lobby.sh` va donc se charger de créer les tubes nommés nécessaires à la communication entre les clients et le programme `game`.

Fonctionnement détaillé

Programme `game`

Le programme `game` est le programme principal du projet. Il va gérer le déroulement de la partie, en gérant les joueurs, les cartes, les têtes de bœufs, etc.

`game.h`

Le fichier `game.h` contient les déclarations des fonctions et des structures utilisées dans le programme `game`. Il contient également les déclarations des variables globales utilisées dans le programme.

Les structures définies dans ce fichier sont :

- `card_t` : structure représentant une carte :
  - `int value` : valeur de la carte ;
  - `int heads` : nombre de têtes de bœufs de la carte ;
- `stack_t` : structure représentant une pile de cartes :
  - `int size` : nombre de cartes dans la pile ;
  - `card_t *cards` : tableau de cartes ;
- `player_t` : structure représentant un joueur :
  - `int id` : identifiant du joueur ;
  - `int score` : score du joueur ;
  - `FILE *in` : fichier d'entrée du joueur ;
  - `FILE *out` : fichier de sortie du joueur ;
  - `stack_t stack` : les cartes que le joueur possède ;

Les variables globales définies dans ce fichier sont :

- `extern int num_players` : le nombre de joueurs ;
- `extern player_t *players` : le tableau des joueurs ;
- `extern stack_t *stacks` : le tableau des piles de cartes ;

`main.c`

Le programme `game` est lancé avec un certain nombre d'arguments, qui vont déterminer le mode de jeu. Le programme va donc commencer par vérifier les arguments passés, initialiser les variables globales nécessaires au bon fonctionnement du programme, puis lancer la fonction `run` qui va gérer le déroulement de la partie.

Le fichier `main.c` contient donc la fonction `main` qui va gérer le déroulement du programme. D'autres fonctions sont définies dans ce fichier afin de déléguer certaines tâches à des fonctions plus petites. Ainsi :

- `static void read_args(int argc, char **argv)` : va lire les arguments passés au programme et va initialiser les variables globales en fonction de ceux-ci ;
- `static void open_fifo(player_t *player)` : va ouvrir les pipes d'entrée et de sortie du joueur passé en paramètre ;
- `static void init_players(void)` : va initialiser les joueurs en fonction des variables globales définies par les arguments passés au programme ;
- `static void init_stacks(void)` : va initialiser les piles de défausse de cartes ;
- `static void bye(void)` : sera appelée à la fin du programme pour libérer toutes les ressources allouées par le programme.

`run.c`

La fonction `run` va gérer le déroulement de la partie. Elle exécutera la boucle de jeu tant que la partie n'est pas terminée. A chaque tour de boucle, elle va distribuer les cartes aux joueurs s'ils n'en ont plus, demander à chaque joueur quelle carte ils souhaitent jouer, définir l'ordre de jeu de la seconde phase à partir du choix précédent puis va faire choisir la pile à chaque joueur et jouer la carte choisie sur la pile choisie.

Le fichier `run.c` contient donc la fonction `run` qui va gérer le déroulement de la partie. D'autres fonctions sont définies dans ce fichier afin de déléguer certaines tâches à des fonctions plus petites. Ainsi :

- `static int game_is_ended(void)` : va vérifier si la partie est terminée en vérifiant si un joueur a atteint le score maximum ;
- `static void get_order(int *order)` : va définir l'ordre de jeu de la seconde phase à partir du choix des joueurs ;
- `static void play_1(void *p)` : va demander au joueur passé en paramètre quelle carte il souhaite jouer et placer la carte choisie à la fin de la pile du joueur ;
- `static void play_2(player_t *player)` : va demander au joueur passé en paramètre quelle pile il souhaite jouer et jouer la carte choisie sur la pile choisie ;
- `static void podium(void)` : va afficher les résultats de la partie à chaque joueur.

`deal_cards.c`

La fonction `deal_cards` va distribuer les cartes aux joueurs à tour de rôle. Elle va distribuer une carte à chaque joueur tant qu'ils n'ont pas le nombre nécessaire de cartes en main. Puis, elle va poser une carte sur chaque pile de défausse.

Le fichier `deal_cards.c` contient donc la fonction `deal_cards` qui va distribuer les cartes aux joueurs. D'autres fonctions sont définies dans ce fichier afin de déléguer certaines tâches à des fonctions plus petites. Ainsi :

- `static int f_heads(int value)` : va calculer le nombre de têtes de bœufs d'une carte à partir de sa valeur ;
- `static void init_deck(stack_t *deck)` : va initialiser le paquet de cartes et le mélanger ;

`play_bot.c`

Ce fichier contient deux fonctions principales : `int play_bot_1(player_t *player)` et `int play_bot_2(player_t *player)`, qui vont respectivement simuler la première et la deuxième phase de jeu (à savoir, le choix d'une carte en première phase et le choix d'une pile en deuxième phase) pour un joueur virtuel. Ces deux fonctions vont appeler des fonctions plus petites afin de déléguer certaines tâches à des fonctions plus petites. Ainsi :

- `static int f_head(stack_t *stack)` : va calculer le score d'une pile de cartes en fonction du nombre de têtes de bœufs qu'elle contient ;
- `static int f_number(stack_t *stack, int id_card)` : va calculer le score d'une pile de cartes en fonction de la différence entre la carte à jouer et la dernière carte de la pile ;
- `static int score(player_t *player, stack_t *stack, int id_card)` : va calculer le score d'une pile de cartes en fonction de la stratégie du joueur virtuel ;

Ces fonctions nous permettront de déterminer la meilleure pile à jouer pour un joueur virtuel en calculant un score pour chaque possibilité de jeu. Le coup qui aura le score le plus faible sera alors joué.

`play_human.c`

Ce fichier contient deux fonctions principales : `int play_human_1(player_t *player)` et `int play_human_2(player_t *player)`, qui vont respectivement demander au joueur humain quelle carte il souhaite jouer et quelle pile il souhaite jouer. Ces deux fonctions vont appeler des fonctions plus petites afin de déléguer certaines tâches à des fonctions plus petites. Ainsi :

- `static void print_cards(FILE *out, card_t *cards, int size, int id)` : va être capable d'afficher un nombre défini de cartes à partir d'un tableau de cartes, avec ou sans identifiant de pile ;
- `static void print_state_1(player_t *player)` : va afficher l'état du jeu pour le joueur humain en première phase ;
- `static void print_state_2(player_t *player)` : fera de même pour le joueur humain en deuxième phase ;
- `static int read_input(FILE *in)` : va lire l'entrée du joueur via le flux correspondant et retourner le choix du joueur ;
- `static int choose(player_t *player, int part)` : va demander au joueur humain quelle carte il souhaite jouer en première phase et quelle pile il souhaite jouer en deuxième phase en vérifiant que le choix du joueur est bien valide.

`debug.c`

Le fichier `debug.c` contient la fonction `debug` qui va afficher l'état détaillé du jeu sur la sortie `stderr`. Elle va appeler des fonctions plus petites afin de déléguer certaines tâches à des fonctions plus petites. Ainsi :

- `static void debug_cards(stack_t * s)` : va afficher les cartes d'une pile ;
- `static void debug_player(player_t *p)` : va afficher l'état détaillé d'un joueur ;
- `static void debug_stacks(void)` : va afficher l'état détaillé des piles de défausse.

Script `build.sh`

Le script `build.sh` va se placer dans le dossier `game` et va compiler le le programme `game` avec `make` selon les directives du fichier `makefile`. Tous les fichiers résultants de la compilation seront placés dans le dossier `bin`. Une fois la compilation terminée, le script se replacera dans le dossier d'origine.

Script `local.sh`

Le script `local.sh` est lui aussi assez simple. Il va faire appel au script `build.sh` pour compiler le programme `game` et va ensuite lancer le programme `game` avec un joueur humain et trois joueurs virtuels.

Script `server.sh`

Le script `server.sh` va tout d'abord faire appel au script `build.sh` pour compiler le programme `game`. Puis, il va lancer en parallèle plusieurs boucles infinies sur chaque port de la plage de ports définie. Chaque boucle lancera le programme `netcat` en écoute sur le port spécifié chaque connexion lancera le script `lobby.sh` qui va gérer les actions demandées par le client. En cas de problème, le serveur peut être arrêté en envoyant un signal `SIGINT` au processus.

Script `client.sh`

Le script `client.sh` est assez simple. En lui passant l'adresse du serveur auquel on souhaite se connecter, il va tenter de se connecter au serveur via `netcat` sur l'un des ports d'une plage de ports définie. En cas d'échec, il va demander à l'utilisateur si ce dernier souhaite se connecter sur un autre port jusqu'à avoir essayé tous les ports de la plage. Une fois connecté, l'utilisateur interagira directement avec le serveur via le script `lobby.sh`.

Script `lobby.sh`

Le script `lobby.sh` est le script le plus complexe de ce projet. Il commence par vérifier que le programme `game` est bien présent et que les dossiers utilisés par le jeu ont bien été créés. Il va ensuite afficher un message de bienvenue à l'utilisateur et le laisser le choix de l'action qu'il souhaite effectuer. Selon le choix de ce dernier, il va faire appel à des fonctions chargées de traiter la demande de l'utilisateur. Ainsi, si l'utilisateur souhaite créer une partie, la fonction `action_create` sera appelée, sinon, si ce dernier souhaite rejoindre une partie, la fonction `action_join` sera appelée. Si le joueur souhaite quitter le jeu, le script cessera son exécution et la connection sera fermée.

Créer une partie

Si l'utilisateur souhaite créer une partie, le script va demander à l'utilisateur de choisir le nom de la partie et le nombre de joueurs de chaque type qu'il souhaite. Si l'entrée de l'utilisateur est correcte, on va créer un dossier du nom de la partie dans le dossier de fonctionnement du script, nous allons ensuite y créer trois fichiers par joueur humain : un fichier `.open` et deux pipes nommés `.in` et `.out` avec comme nom un identifiant de joueur. Le fichier `.open`, par sa présence, indique qu'aucun autre joueur ne s'est connecté à la partie avec l'identifiant spécifié et indique donc que la place est libre. Enfin, le script lance le programme `game` avec les paramètres définis par le client.

Rejoindre une partie

Si l'utilisateur souhaite rejoindre une partie, le script va demander à l'utilisateur d'entrer le nom de la partie qu'il souhaite rejoindre. Si la partie existe et possède un emplacement libre, le script va supprimer le fichier `.open` du joueur et va connecter le client aux pipes nommés du joueur, pouvant alors interagir avec la partie via ces pipes.