Atomic Teddy Investors

Auteurs

Saad Zoubairi **Quentin Lamour** **Guillaume Mannone**

Table des matières

1 Introduction

1.1 Contexte

L'objectif de ce rapport est de décrire la réalisation du 1er projet d'algorithmique et de programmation par des étudiants de 1ère année à l'ENSEIRB-MATMECA, en filière informatique. Ce projet, intitulé *Atomic Teddy Investors*, a été réalisé en langage C - Norme C99. Les étudiants devaient concevoir une version de base (Achievement 0) avec certaines structures de données imposées, puis pouvaient réaliser différents Achievement débloqués au fur et à mesure du projet.

1.2 Description du sujet

Des ours (les *Teddy*) ont envahi la Sicile et se sont intéressés aux comportements humains. Ils se sont passionnés pour l'économie de marché, et sont particulièrement attirés par le miel qui leur sert de monnaie (*Honey* en anglais).
Le sujet a pour but de mettre en concurrence les *Teddy* avec différentes transactions sur des places d'échanges (*stockex* en anglais). Les ours jouent chacun leur tour, réalisent une transaction de manière aléatoire ou avec une certaine stratégie, puis retournent dans la file d'attente. Les ours passent du temps à échanger leurs ressources avec la place d'échange et le plus rapide joue en premier. A la fin de la partie, l'ours ayant le plus de ressources en *equivalent Honey* a gagné la partie. L'*equivalent Honey* est la conversion des ressources d'un ours en leur valeur de référence en *Honey*.

1.3 Contraintes et choix

Dans le projet, nous avons choisi de laisser les 3 ressources proposées par le sujet : le miel (*Honey*), la clé à molette (*Monkey Wrench*) et le Moteur de Hors-Bord (*Outboard Motor*). Nous avons également fait le choix de définir 3 places d'échange. Sur chacune d'elles sont disponibles 3 transactions choisies aléatoirement. Celles-ci proposent un échange de ressources qui peut être favorable ou non pour le *Teddy* (exemple : acheter une ressource pour 10 *Honey* alors qu'elle en vaut 100). Cependant, il n'est pas possible de recevoir une ressource si elle est également donné par le joueur dans la même transaction (exemple : échanger 10 *Honey* et 1 clé à molette contre 3 clés à molette). De plus, le sujet impose un nombre de 20 joueurs et de 20 ressources différentes maximum, un nombre de 100 places d'échange maximum et de 100 transactions maximum par place d'échange.

2 Méthode de réalisation

2.1 Outils utilisés

Les outils d'aide au développement que nous avons utilisés nous ont été très fortement recommandé par nos encadrants de projet.
Nous avions accès à la forge de l'ENSEIRB-MATMECA pour sauvegarder notre avancée et avoir accès à toutes les versions de notre code, à l'aide du logiciel de gestion de version *Git*. Cela nous a permis d'avancer séparément sur le projet, que ce soit chez nous ou lors des séances de travail, sans la contrainte du stockage de notre progression.
Nous avons utilisé l'éditeur de texte Emacs pour la conception du code et ainsi se familiariser un peu plus avec celui-ci.
Enfin, ce rapport a été rédigé en LaTeX, les documents relatifs à sa conception son disponible sur la forge de notre projet dans la branche *Rapport*.

2.2 Structure des fichiers

2.2.1 Arborescence du répertoire

Le projet est rangé de la sorte. A la racine du dossier, nous avons un fichier *README.txt* explicitant le rôle de chaque fichier, un fichier *authors.txt* qui contient le nom des auteurs du projet et leur contact ainsi qu'un dossier *Rapport* qui contient les fichiers nécessaires à la création de ce rapport. Nous avons également le dossier *latest* qui contient la dernière version du projet, puis différents dossiers *achiev ?* (jusqu'à achiev5 au maximum) qui contiennent la version du code validant l'achievement concerné.

Chacun des dossiers *achiev ?* et le dossier *latest* contiennent un fichier *Makefile*, un dossier *src* qui contient les fichiers sources du projet, et un dossier *tst* qui contient les fichiers tests du projet.

2.2.2 Liens entre les fichiers .c et .h

Nous avions deux fichiers imposés pour la réalisation de ce projet : *good.c* et *stockex.c* qui décrivent les ressources et les places d'échanges. Nous avons décidé de créer un fichier *queue.c* qui décrit le système de file de priorité du jeu. De plus, nous avons réparti les fonctions de jeu des *Teddy* sur 2 fichiers : *teddy.c* et *strategy.c*. Le premier contient toutes les fonctions auxiliaires nécessaires au bon fonctionnement du second, qui lui contient les différentes fonctions *play* selon les stratégies assignées aux *Teddy*. Tous ces fichiers sont accompagnés de leurs homonymes en .h pour permettre l'appel de certaines fonctions dans plusieurs fichiers .c lorsque cela est nécessaire. Nous avons également un fichier *project.c* qui exécute la boucle générale du jeu. Nous avons réalisé un schéma simplifié qui explicite les liens entre nos fichiers .c et .h.
Grâce à l'outil qu'est le Makefile nous avons pu exécuter notre programme principale en utilisant la commande *make*. Aussi, nous avons utilisé la commande *make test* qui permet d'afficher les résultats de nos fonctions tests. Enfin, nous nous sommes intéressés à définir une commande *make clean* qui permet de supprimer les exécutables ainsi que les fichiers objets (.o).

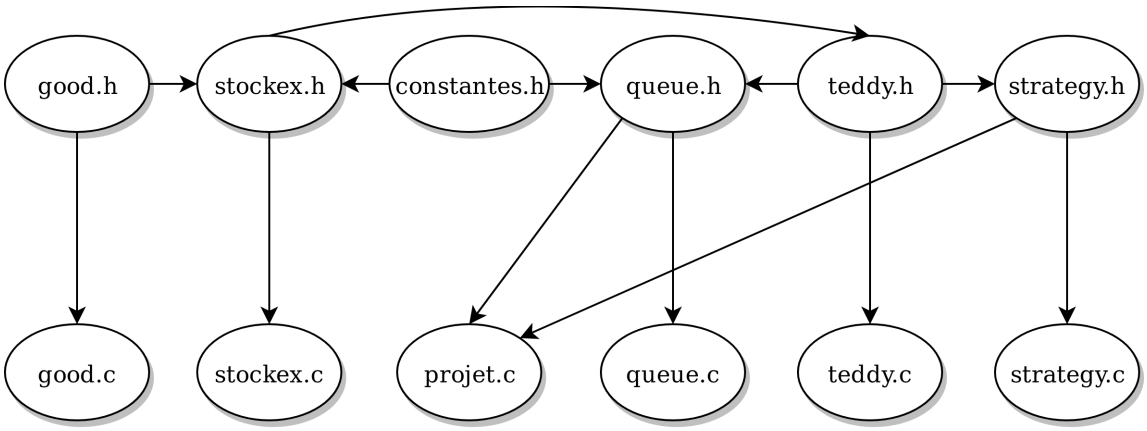

FIGURE 1 – Liens entre les fichiers

2.3 Structure des données

Afin de répondre au mieux au problème posé, nous avions comme obligation de créer des structures et/ou des énumérations. Nous allons donc présenter succinctement nos choix de résolution.

2.3.1 Définition des constantes

Pour faciliter les modifications en cas de besoin, nous avons créer un fichier constantes.h qui regroupe les définitions de nos constantes. Celles-ci ont pour but de permettre un changement rapide du paramétrage du jeu. Par exemple, si l'on veut changer le nombre maximum de *Teddy* dans une partie, il suffit de le changer dans ce fichier au lieu de changer ce nombre partout où il est impliqué (comme la taille de la *queue*).
Les constantes définies dans ce fichier sont : *MAX_CHAR* fixée à 64 et qui représente la taille maximale d'une chaîne de caractère. *MAX_TRANSACTION* fixée à 20 et représente le maximum de transactions qu'une place d'échange peut contenir. *MAX_TEDDIES* est égale à 20 et indique la limite des joueurs qui peuvent participer au jeu. *SET_TIME* est une constante fixée à 1000 et indique le temps de jeu à utiliser par défaut s'il n'est pas indiqué au début du jeu.

2.3.2 Description et stockage des ressources

La création des ressources et du portefeuille pour réaliser des transactions ont été imposées dans le sujet. Il était possible cependant de rajouter des ressources dans l'énumération de celles-ci mais nous avons décider d'en laisser trois par soucis de simplification du problème. La première ressource *Honey* est initialisée à 0 de manière à s'assurer que l'énumération commence bien avec une valeur de 0. Ainsi, la constante *MAX__GOOD* vaut bien 3 comme le nombre de ressources définies.

La structure *wallet* n'a pas été changé puisqu'elle décrit déjà correctement le problème. Elle contient un tableau d'entiers, nommé *data* correspondant à la quantité de chaque ressource, classé dans le même ordre que l'énumération des ressources (en premier la quantité de *Honey*, etc...).

```
enum good {
            HONEY = 0,
            MONKEY__WRENCH,
            OUTBOARD__MOTOR,
            MAX__GOOD,
            ERROR__GOOD = -1,
};
```

2.3.3 Places d'échange et transactions

Dans un premier temps, nous nous sommes placés du point de vue de la place d'échange afin de définir une transaction. Pour cela, nous avons modélisé la transaction comme deux tableaux de type *struct wallet*, c'est-à-dire deux tableaux contenant chacun un portefeuille. Ainsi, les portefeuilles sont en fait ce que la place d'échange va donner au *Teddy* et ce que le *Teddy* va dépenser. De plus, la structure contient le numéro de la place d'échange actuelle où la transaction est faite : *actual_stockex*, ainsi que le numéro de la prochaine place d'échange où la transaction mène : *next_stockex*. En effet, nous avons défini un tableau qui contient les places d'échange. Ainsi, ces deux entiers représentent les indices des places d'échange dans le tableau.
Les places d'échange possèdent trois informations : *name* contenant son nom, *num_transactions* contenant le nombre de transactions disponibles sur cette place, et *transactions*. Ce dernier est un tableau contenant des éléments de type *struct transac*, c'est-à-dire qu'il contient les transactions qui peuvent s'effectuer ici.

```
struct transac {
  struct wallet tableau_in;
  struct wallet tableau_out;
  int actual_stockex;
  int next_stockex;
};
```

```
struct stockex {
  char name[MAX_CHAR];
  int num_transactions;
  struct transac transactions[MAX_TRANSACTION];
};
```

2.3.4 File de priorité

Pour définir la file de priorité des *Teddy*, la *struct queue* contient une variable *taille* contenant la taille actuelle de la *queue*. De plus, elle contient un tableau *file*, rempli de *Teddy*, qui permet de faciliter le déroulement du jeu, dont nous détaillons la structure plus bas.

2.3.5 Définition des *Teddy*

La structure permettant de décrire les *Teddy* contient un portefeuille décrit à l'aide de la *struct wallet* définie précédemment. Ils sont également décrits par une variable *time* qui contient leur temps de jeu actuel et par une variable *id* qui contient leur numéro pour les identifier (ce numéro varie de 1 à $MAX\_\_TEDDIES$). La structure contient un élément qui représente la stratégie suivie par le *Teddy* lors de la réalisation des transactions, ainsi qu'une variable *stockex* qui contient la place d'échange actuelle du joueur. De plus, notre représentation de la structure contient un tableau contenant des chaînes de caractères qui indiquent les noms des places d'échanges déjà visitées par le *Teddy*, ainsi qu'un entier *num_stockex* informant sur la taille de ce tableau.

```c
struct teddy {
  struct wallet portefeuille;
  int time;
  int id;
  int fun_play;
  const struct stockex *stockex;
  char* visited_stockex[MAX_STOCKEX];
  size_t num_stockex;
};
```

3 Présentation des résultats

Nous avons programmer un code qui fait jouer en compétition un certain nombre de *Teddy*. Chaque *Teddy* a un portefeuille de ressources et échange à chaque tour de jeu avec une place d'échange à travers une transaction. Chaque place d'échange possède ses propres transactions et en réaliser une permet de changer de place d'échange. Les participants sont organisés à travers une file de priorité qui est ordonnée afin de faciliter la connaissance du tour de chaque joueur. Certains *Teddy* ont une stratégie de jeu que nous avons prédéfinie et qui les fera jouer d'une certaine manière.

3.1 Paramètres optionnels du jeu

Avant de commencer notre programmation de la boucle principale, nous avons essayé de définir une fonction qui permet de récupérer les options proposées lors de l'exécution de notre programme ou retourne un message d'erreur si les options ne sont pas valides.

Nous avons proposé aussi une vérification des valeurs des options afin qu'elles soient conformes aux limites proposées par le sujet. Ainsi nous avions à définir trois options principales.

3.1.1 Graine aléatoire

La graine aléatoire permet d'associer la génération d'une partie à un numéro. Cela permet de rejouer plusieurs fois une même partie avec la même génération de nombres aléatoires. Cela peut être utile pour rejouer des scénarios où l'on aurait trouvé une ou plusieurs erreurs.

3.1.2 Nombre de joueurs

Grâce à cette option, l'utilisateur a le droit de choisir le nombre de participants au jeu en ajoutant la commande "-n N", avec N un entier naturel compris entre 0 et *MAX_ TEDDIES*. Par conséquent, en cas de choix d'un nombre n'appartenant pas à cet ensemble, le jeu sera annulé grâce aux moyens de vérification. Cette option prend par défaut la valeur 2 si l'utilisateur n'insère aucune valeur.

3.1.3 Temps de jeu

Cette option permet de définir le temps maximal de jeu de la partie. Elle est accessible grâce à la commande "-m M", avec M un entier naturel strictement positif. Le programme s'arrête en cas de choix d'un entier nul ou négatif et affiche un message d'erreur. Cette option prend par défaut la valeur 1000 si l'utilisateur n'insère aucune valeur.

3.2 Processus de transaction

Dans le but de reprendre les différents métiers exercés par la race humaine, les ours ont décidé de recopier le principe d'économie du marché afin de subvenir à leurs besoins. Contrairement aux humains, ils ont décidé de considérer le miel (*Honey*) comme leur valeur de référence et ainsi comme leur ressource principale. Ce principe économique crée un défi entre différents ours qui essayent de se montrer plus forts en ayant le plus de ressources possible.

Le jeu s'intéresse principalement à la réalisation de transactions par chaque joueur lors d'un tour de jeu. Ainsi, il était nécessaire de définir tout d'abord nos transactions qui sont regroupées

dans différentes places d'échange. Ces dernières restent invariantes tout le long de jeu et sont primordiales pour le bon déroulement du jeu.

Comme défini précédemment, le principe d'une transaction est un portefeuille qui contient ce que le joueur devrait vendre ainsi qu'un autre portefeuille qui contient ce qu'il devrait recevoir en échange. En effet, ces transactions sont le seul moyen d'échange pour les *Teddy*. Elles permettent soit d'augmenter leur capital en leur proposant des offres avantageuses, soit de diminuer leur capital en leur proposant des offres désavantageuses. Enfin, elles permettent au *Teddy* de se déplacer. La transaction contient le numéro de la place d'échange où le *Teddy* jouera au prochain tour s'il réalise cette transaction.

3.3 File de priorité

Il est évident que chaque ours cherchera à être le plus fort et gagner le plus de ressources possibles, ce qui pourrait engendrer des conflits. Ainsi, afin d'organiser le déroulement d'une partie et de rendre équitable l'accordement des tours de jeu aux joueurs, nous avons implémenté une file de priorité. Celle-ci contient au début du jeu les joueurs participants. Elle est configurée de façon à être réorganisée à chaque tour de jeu dans le but de classer les joueurs par ordre croissant de temps passé dans la place d'échange.

En effet, le premier *Teddy* dans la file est censé être le joueur prioritaire puisqu'il a le temps de jeu le plus bas. La manipulation de la file se fait à travers plusieurs fonctionnalités qui font retourner le *Teddy* prioritaire grâce à la fonction *queue__pop* et le réintègre dans la file après son tour de jeu. Cette insertion se fait à travers une comparaison du temps du *Teddy* avec celui des autres joueurs, la fonction qui réalise ce travail est appelée *queue__push*.

```
void queue__push(struct queue* h, int priority, struct teddy* t);
struct teddy* queue__top(struct queue* h);
struct teddy* queue__pop(struct queue* h);
```

FIGURE 2 – Fonctions interagissant avec la *queue*

Le principe de file de priorité est efficace dans notre cas puisqu'il simplifie la structure et le déroulement du jeu. En effet, elle comporte une composante qui représente sa taille et qui permet donc de connaître le nombre de joueurs présents dans la place d'échange à tout moment du jeu.

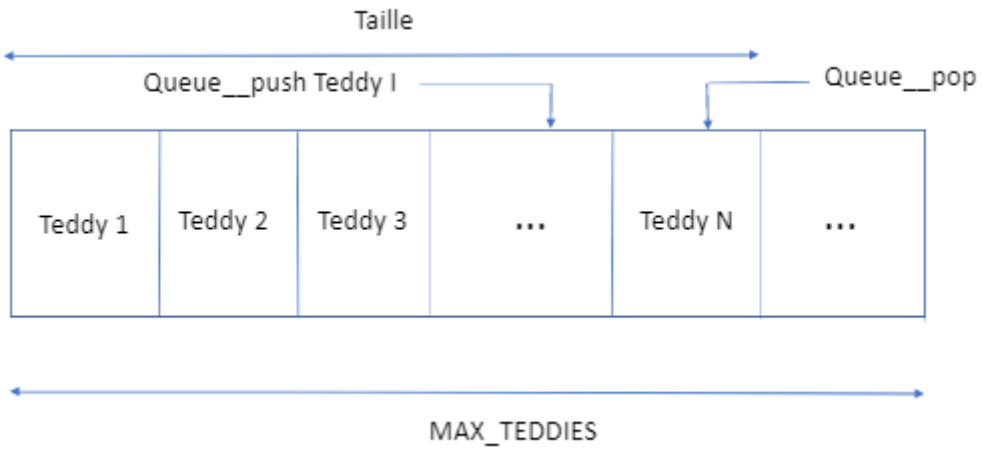

FIGURE 3 – Représentation d'une file de priorité

3.4 Stratégies de jeu

Afin de rendre le jeu plus intéressant, nous avons essayé de définir des stratégies qui vont permettre d'optimiser les gains. Ainsi, grâce à ces stratégies, à chaque place d'échange, le joueur sait quelle transaction il va réaliser. De ce fait, nous avons essayé de définir trois stratégies :

3.4.1 Stratégie 0

Le jeu est effectué aléatoirement, la transaction effectuée par le *Teddy* lui permet soit d'augmenter ou diminuer la valeur de son portefeuille. Ainsi, si la transaction choisie est possible, il la réalise sinon il passe son tour pendant une unité de temps. Ainsi, cette stratégie représente le déroulement d'un tour de jeu pour la version de base du jeu. Par conséquent, un *Teddy* sans stratégie est un joueur qui suit la stratégie 0.

3.4.2 Stratégie 1

Le joueur cherche à réaliser la transaction qui lui fait gagner le plus de *Honey*. Ainsi, nous avons eu à définir une fonction qui calcule le gain d'une transaction donnée. Cela va nous permettre de l'appliquer pour toutes les transactions disponibles dans la place d'échange actuelle, et de comparer les résultats afin de choisir le gain maximal. Par suite, si toutes les transactions sont perdantes, alors la transaction choisie sera celle qui fait perdre le moins. Une condition primordiale impose que le *Teddy* doit être capable de réaliser la transaction afin qu'elle soit valide. En effet, cette stratégie est optimale mais ne donne aucune information sur la prochaine place d'échange. Par conséquent, une transaction peut être la meilleure et mener vers une place d'échange perdante.

3.4.3 Stratégie 2

Avec de cette stratégie, nous avons cherché à analyser la transaction qui mène vers la place d'échange qui a la plus grande valeur moyenne de gain. Par suite, nous avons défini une fonction qui permet de calculer la valeur moyenne de gain d'une place d'échange. Grâce aux résultats de cette fonction, le joueur va choisir la transaction qui le mène vers la place d'échange avec le gain maximal. Aussi, le *Teddy* doit être capable de réaliser la transaction sinon elle n'est pas prise en compte. Cette stratégie se présente optimale mais peut parfois faire que le *Teddy* effectue une transaction perdante si elle mène vers la place d'échange avec le plus grand gain moyen.

3.4.4 Stratégie 3

Il s'agit d'une stratégie moins gagnante que les deux premières. Le *Teddy* cherche à choisir une transaction positive parmi celles possibles. Ainsi, il choisit une transaction avec un gain positif aléatoirement. Si le joueur ne peut effectuer aucune transaction positive, il choisit aléatoirement parmi les transactions disponibles. Le but de cette stratégie est de pousser le *Teddy* à ne choisir que des transactions positives.

Ainsi le choix de réalisation d'une transaction selon la stratégie peut se faire comme suit sachant que la stratégie 0 permet de choisir parmi toutes les transactions donc nous n'avons pas eu à la modéliser :

3.5 Déroulement du jeu

D'abord, nous avons tâché à créer la file de priorité et insérer les participants par ordre croissant des identifiants, puisque leur temps est nul au début du jeu. Le code qui réalise cette fonctionnalité est le suivant :

Le tableau utilisé regroupe les *MAX_ TEDDIES* qui peuvent jouer une partie. Grâce aux identifiants et à la variable optionnelle **players**, nous allons réaliser l'insertion.
Ensuite, le principe du jeu porte sur l'enchaînement de tours de jeu tant que le temps global est inférieur au temps maximal. Ainsi, à chaque tour, le *Teddy* prioritaire est retiré de la file de priorité et effectue des transactions qui font changer son portefeuille et son temps de jeu.

3.5.1 Tour de jeu d'un *Teddy*

A chaque tour de jeu, un *Teddy* est retiré de la file de priorité pour jouer et ainsi réaliser une transaction. Selon sa stratégie et son portefeuille actuel, le *Teddy* choisit une transaction. Ensuite, une fonction vérifie que le *Teddy* n'a pas essayé de tricher en effectuant une transaction qui n'est pas

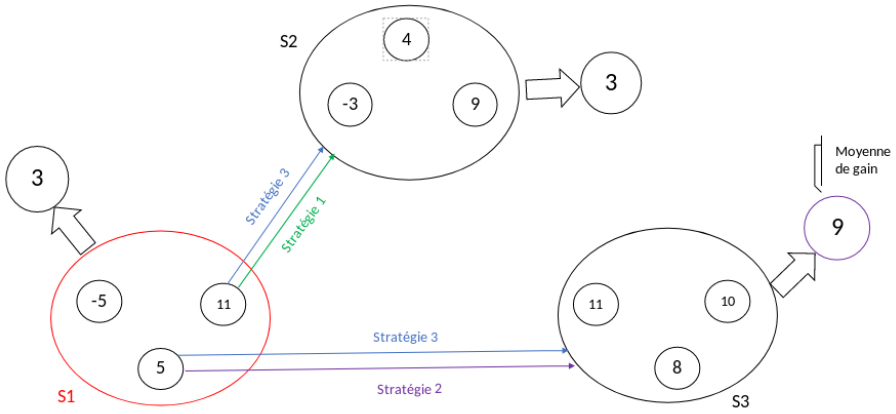

FIGURE 4 – Choix de transaction selon les stratégies

```
for (int i=0; i<players; i++)
   queue__push(&queue, 0,&teddies[i]);
```

FIGURE 5 – Remplissage de la file de priorité avant le début du jeu

disponible sur sa place actuelle. Une autre fonction simule la transaction pour vérifier qu'elle ne va pas créer un *overflow*. En effet, les éléments du portefeuille étant de type *unsigned int*, l'*equivalent Honey* d'un *Teddy* est un *int*. Ainsi, si la transaction augmente son portefeuille au point de dépasser la valeur maximale supportée par un *int*, cela va créer un *overflow* et donc fausser la partie. S'il n'a pas triché, le *Teddy* réalise donc la transaction choisie précédemment, puis se déplace vers la place d'échange associée à celle-ci. Son temps de jeu est alors incrémenté de 1 en le remettant dans la *queue*. Cela est dû au fait que les *Teddy* ne réalise qu'une seule transaction dans la version finale du projet.

```
    struct teddy* active_teddy = queue__pop(&queue);
    int index_transac = array_play[active_teddy->fun_play](active_teddy);
    if (index_transac == -1)
        queue__push(&queue, active_teddy->time + 1, active_teddy);
     else
        {
        int check = check_overflow(active_teddy, index_transac);
        if (active_teddy->stockex ==
            transac__stockex(stockex__transaction(active_teddy->stockex,
                                                  index_transac)))
            {
                if (check == 1)
                    realise_transaction(active_teddy, index_transac);
                queue__push(&queue, active_teddy->time + 1, active_teddy);
            }
        }
```

FIGURE 6 – Tour de jeu d'un *Teddy* dans la boucle principale

Pour simplifier la résolution du problème, nous avons créé de nombreuses fonctions auxiliaires. **array_play** est un tableau contenant des pointeurs vers les fonctions *play* des joueurs. Chaque joueur suit une stratégie qui lui est affectée aléatoirement. Cette stratégie représente donc l'indice de la fonction *play* dans le tableau. En effet, grâce à cet appel, nous affectons au joueur l'index de la transaction qu'il va réaliser.
check_overflow est une fonction qui vérifie si la valeur du portefeuille du joueur ne réalise pas

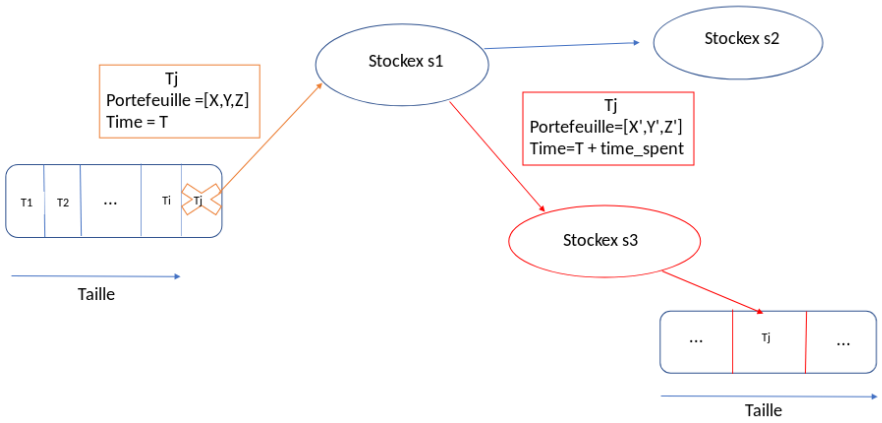

FIGURE 7 – Simulation d'un tour de jeu

un overflow. Cette fonction effectue une simulation du déroulement de la transaction et calcule la valeur du portefeuille du *Teddy*. Ainsi, si la valeur calculée est négative, le joueur ne réalisera pas cette transaction et passera son tour, sinon la condition est validée et ainsi la boucle de jeu testera les autres conditions.
realise_transaction : si le *Teddy* peut effectuer la transaction alors la fonction effectue l'échange entre le joueur et la place d'échange. Cette fonction est appliquée pour toutes les ressources affectées par cet échange. Ensuite, elle affecte au *Teddy* la prochaine place d'échange où il va jouer.

En cas de jeu du *Teddy* la variable *time_spent*, qui contient le temps du jeu du *Teddy*, est calculée à travers une relation mathématique :

$$time\_spent = \lceil 1 + \log_2(h) \rceil \tag{1}$$

La variable "h" représente le nombre de transactions qu'a réalisé le *Teddy*. Ainsi, comme les *Teddy* ne réalisent qu'une transaction dans cette version du projet, le temps de jeu est toujours incrémenté de 1. Nous avons utilisé des fonctions prédéfinies dans la bibliothèque *math.h* comme **log2** qui calcule le logarithme binaire d'un entier strictement positif, et la fonction **ceil** qui nous a permis d'obtenir la partie entière supérieure du temps de jeu qui est un entier naturel.

3.5.2 Affichage des résultats

A la fin du jeu, nous avons présenté le résultat de chaque participant sous forme de tableau, en affichant la valeur de son portefeuille en équivalent *Honey*. En comparant les résultats de tous les participants, nous affichons le *Teddy* gagnant et la valeur de son portefeuille. Le joueur gagnant est celui qui a réalisé les meilleurs transactions et qui a pu collecter le plus de ressources. Les résultats sont calculés grâce aux valeurs que nous avons attribués aux ressources en *Honey*.

$$Result = \sum_{i=0}^{MAX\_GOOD} (good\_value(i) * portefeuille\_teddy[i]) \tag{2}$$

```
void parse_opts(int argc, char* argv[]);
void check_opts(int argc, char* argv[]);
void display_results();
```

FIGURE 8 – Récupération et vérification des paramètres optionnels et affichage des résultats

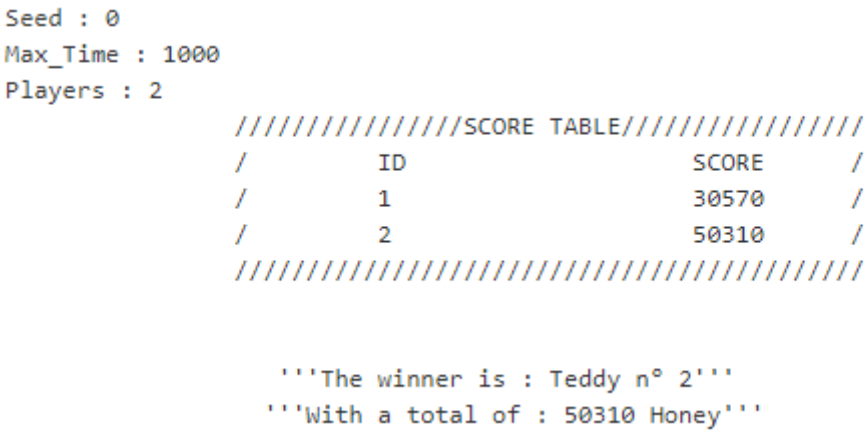

FIGURE 9 – Résultat d'exécution

3.6 Tests de validité

Pour vérifier la correction et la terminaison de nos fonctions, nous avons choisi de les tester individuellement avec des exemples simples et concrets.

3.6.1 Structures

Pour tester nos structures, nous avons créer les fichiers tests *test_good.c* et *test_stockex.c*. Nous avons vérifié toutes les fonctions de *good.c* et *stockex.c* à travers une mise en situation simplifiée du projet. En effet, nous avons créer les ressources, une place d'échange et des transactions pour s'assurer qu'elles retournaient bien ce que nous voulions.

3.6.2 File de priorité

Dans le fichier *test_queue.c*, nous avons dû définir des *Teddy* pour simuler une file de priorité. Il est important de vérifier le bon fonctionnement des fonctions *queue__push* et *queue__pop* qui sont essentielles au déroulement du jeu. Cela nous a permis de vérifier que la file de priorité fonctionnait correctement.

3.6.3 Tour de jeu

Dans le fichier *test_teddy.c*, nous avons créer un *Teddy* pour tester son initialisation et vérifier le bon fonctionnement d'un tour de jeu quelconque. De plus, nous avons aussi testé les fonctions auxiliaires définies dans le fichier *teddy.c* dans des versions précédentes du projet pour être certain des résultats retournés. Enfin, dans un fichier *test_strategy.c*, nous avons effectué plusieurs manipulations autour de *Teddy* avec différentes stratégies. Ainsi, nous avons pu vérifier que les *Teddy* agissaient différemment selon la stratégie qui leur était affectée.

4 Complexité et discussion

4.1 Complexité

De manière générale, la complexité en temps et en espace du code dépend de celles des fonctions *play*, *queue__push* et *queue__pop* qui sont appelées à chaque tour de jeu. La complexité en temps et en espace de la fonction *queue__pop* et de chaque fonction auxiliaire à la fonction *play* est constante. Ainsi, nous pouvons affirmer que la fonction *play* est de complexité constante. De plus, la fonction *queue__push* a une complexité linéaire en fonction de la taille de la *queue*. La boucle principale est de complexité linéaire en fonction du nombre de tours. De ce fait, la complexité globale de notre programme est linéaire selon deux variables : le nombre de *Teddy* dans la *queue* et le nombre de tours de la partie.

4.2 Problèmes rencontrés et solutions apportées

Lors de la réalisation du projet, nous avons été confrontés à plusieurs problèmes de résolution. Lors de l'initialisation des *Teddy*, nous les avions créés de manière non optimisée dans un tableau afin de les insérer dans la file de priorité lors du premier tour. En solution à cela, nous avons créé une fonction *init_ _teddy* qui crée les *Teddy* dans le tableau à l'aide d'une boucle *for*. Cela permet d'alléger le code et de rester logique dans sa structuration.
De plus, en essayant de définir la fonction *play*, nous nous sommes vite rendus compte de la nécessité de diviser le problème en plusieurs sous-problèmes. Nous avons donc réalisé plusieurs fonctions auxiliaires pour répondre aux besoins de cette fonction.
Ainsi, ces fonctions nous ont parfois causé quelques problèmes. Par exemple, la fonction *max_ transac* pouvait prendre trop de temps à s'exécuter quand le *Teddy* avait beaucoup de ressources. En effet, notre algorithme vérifiait si le *Teddy* pouvait faire une fois la transaction, puis deux fois et ainsi de suite. Par conséquent, nous avons mis en place une boucle qui parcourt les types de ressources et retourne directement le maximum à l'aide d'une division euclidienne.

4.3 Améliorations possibles

Notre code présente plusieurs parties qui pourraient être optimisées. D'une part, nous utilisons beaucoup de tableaux associés à une variable représentant leur taille actuelle. Ainsi, cette modélisation peut engendrer des coûts en mémoire et en temps d'exécution importants, même si cela n'a pas trop d'impact à notre échelle. D'autre part, nous avons créé beaucoup de variables pour solutionner le problème ce qui peut parfois rendre difficile la lecture de notre code par autrui. De plus, malgré notre investissement dans l'optimisation de l'arborescence et de la structure de nos fichiers, celles-ci peuvent sûrement être encore améliorées. Enfin, nous avons créé de nombreuses auxiliaires notamment dans le fichier *teddy.c* pour la modélisation du tour de jeu des *Teddy*. Celles-ci effectuent des boucles *for, while, if...* Par conséquent, nous avons une complexité cyclomatique qui n'est pas optimale.

5 Conclusion

En conclusion, la problématique du sujet était de réussir à produire un code capable de simuler un jeu avec des contraintes. Il fallait que l'utilisateur puisse donner un nombre de joueurs souhaités, un temps maximal souhaité et optionnellement un graine de jeu pour sauvegarder sa partie. Concernant la simulation du jeu, certaines fonctions et structures nous ont été imposées pour permettre le remplacement de nos fichiers par d'autres. Cela permet de pouvoir apporter des changements ciblés au code sans avoir à tout changer manuellement. Pour cela, nous avons défini à notre manière les structures représentant les transactions, les places d'échange, la file de priorité et les ours en fonction des problèmes que nous rencontrions. De plus, nous avons créé des fonctions auxiliaires pour alléger nos fonctions telles que *play0* ou *realise_ transaction* qui ont besoin d'une grande quantité d'information. Nous avons également choisi de répartir le code dans plusieurs fichiers .c différents en fonction de ce qu'ils allaient contenir comme partie du code. Enfin, nous avons essayé d'optimiser au maximum les inclusions de nos fichiers entre eux.

6 Références

Une liste des principales sources et références utilisées pour ce projet.

6.1 Ressources internes

— https ://www.labri.fr/perso/fmoranda/pg101/
— https ://www.labri.fr/perso/renault/working/teaching/projets/
— https ://moodle.bordeaux-inp.fr/
— https ://thor.enseirb-matmeca.fr/

6.2 Ressources externes

— https ://stackoverflow.com/
— https ://openclassrooms.com/
— https ://fr.wikibooks.org/wiki/LaTeX
— https ://www.developpez.net