Atomic Teddy Investors

Rapport de projet

Projet de programmation - S5

– I1 - ENSEIRB-MATMECA –

Encadrants : M. David RENAULT - M. Michaël CLEMENT

Par Ismail ELOMARI ALAOUI - Hugo BERNARD–HÊME

15 septembre 2021

Table des matières

Introduction

Contexte

Ce rapport a pour objectif de décrire l'ensemble du processus de réalisation du premier projet de programmation des élèves en première année d'informatique à l'ENSEIRB-MATMECA. Le projet consiste en l'implémentation d'un jeu appelé ATOMIC TEDDY INVESTORS, en langage C. Nous devions, dans un premier temps, établir une version de base selon des consignes précises, puis réaliser différents *Achievements* dévoilés en cours de réalisation du projet.

Description du sujet

Ce projet nous amène dans un monde parallèle dans lequel les ours, que l'on appellera des "*teddies*", ont renversé les humains et dominent donc la terre. Ils y développent alors une nouvelle économie, et notre objectif sera d'émuler, sous la forme d'un jeu, toutes les transactions qui forment cette économie. Plusieurs versions du jeu seront à développer, qui présentent chacune quelques détails particuliers. Ces versions du jeu seront notées, dans toute la suite, "*Achievement*" suivi d'un numéro (par exemple *Achievement* 0 pour la version de base du jeu).
Nous avons développé les *Achievements* 0 à 3, soit tous les *Achievements* possibles. Ce rapport se concentrera ces quatre versions du jeu. Nous y détaillerons les algorithmes utilisés, les complexités de ceux ci, ainsi que les raisons des choix algorithmiques effectués.
Voici, de façon relativement succinctes, les consignes principales à respecter concernant ces *Achievements* :

— *Achievement* 0 :

Ce premier *Achievement* étant la création du noyau du jeu, il nous fallait créer toutes les structures permettant de générer des transactions entre des *teddies* et une place d'échange, sous forme de troc. Nous devions aussi implémenter la boucle principale de jeu, qui fait jouer tour à tour un certain nombre de *teddies* pendant un nombre de tours impartis. A chaque tour, nous devions faire choisir aléatoirement une transaction possible à un *teddy*, lui même choisi selon sa priorité dans une file de priorité, qui si il possède suffisamment de ressources réalise un nombre de fois aléatoire cette transaction. Finalement, notre programme doit afficher le *teddy* gagnant, c'est à dire le *teddy* dont la valeurs marchande de toutes ses ressources, calculée en miel (la "monnaie" des ours), est la plus élevée à l'issue du jeu.

— *Achievement* 1 :

Cet *Achievement* n'est ni plus ni moins qu'une extension de l'*Achievement* 0. Ici, plusieurs places d'échanges sont accessibles aux *teddies*, que nous devions implémenter dans le code. Néanmoins, avant d'y accéder, ceux-ci doivent les débloquer en effectuant des transactions particulières. La si célèbre curiosité des ours les poussant à explorer un maximum de places d'échanges, le choix d'une transaction par un *teddy* ne sera plus nécessairement aléatoire. En effet, nous devions faire en sorte que si certaines transactions permettent à un *teddy* de débloquer une place d'échange où il n'est pas encore allé, il doit choisir aléatoirement seulement parmi ces transactions.

— *Achievement* 2 :

Cet *Achievement* demande l'implémentations de différentes stratégies que les *teddies* peuvent utiliser afin

d'optimiser leurs chances de gagner.

— *Achievement* 3 : Dans ce dernier *Achievement*, le notion d'offre et de demande est à prendre en compte. Si une ressource est rare dans l'ensemble des marchés, sa valeur augmente. A l'inverse, sa valeur diminue si elle est présente en excès dans les places d'échanges.

Problématiques

Ce projet nous amène, au fil de ses *Achievements*, à poser plusieurs problématiques algorithmiques, que nous avons du résoudre afin de le mener à bien. En voici une liste des principales :

— De quelle manière implémenter une liste de priorité, qui soit au moins fonctionnelle et idéalement qui permette à ses fonctions de manipulation de base d'être efficace en terme de complexité ?
— Comment laisser, en option, l'utilisateur choisir quelques paramètres du jeu (par exemple le nombre de joueurs) ?
— Comment implémenter les places d'échanges et les transactions de façon à ce que les *teddies* puissent naviguer entre chaque place en respectant les règles définies ?
— Quelles stratégies peuvent être implémenter afin d'optimiser les chances de victoire de quelques *teddies* ? Comment peut-on les implémenter en prenant en considération des complexités qui peuvent être très sévères ?
— De quelle manière peut-on mémoriser efficacement les ressources de tous les *teddies* ?

1 Méthode de travail

1.1 Répartition des taches

Tout d'abord, notons que ce projet a été du début à la fin un travail d'équipe complet, dans lequel nous avons appris tout du long à beaucoup échanger et à travailler ensemble, tout en se répartissant les taches quand cela était nécessaire.

1.2 Thor Project

Nous avons utilisé la Forge (le serveur *ThorProject*) de l'ENSEIRB-MATMECA pour gérer les différentes versions de notre projet. L'évolution du projet visible sur cette Forge peut mettre en évidence nos progrès en tant que débutants programmeurs.

1.3 Emacs - LaTeX- Ti*k*Z

Pendant l'ensemble de ce projet, nous avons utilisé l'éditeur de texte Emacs pour toute la programmation C, afin de nous familiariser avec toutes ses différentes commandes, touches, macros et raccourcis.
Nous avons utilisé comme il nous l'avait été demandé LaTeX pour rédiger ce rapport final. Afin de simplifier le travail simultané sur l'écriture de ce rapport, nous avons utiliser *Overleaf*, un éditeur LaTeX en ligne.
En outre, tous les graphes de ce rapport sont réalisés avec le pack Ti*k*Z, dans le but de nous familiariser avec ce pacage de LaTeX.

1.4 *Makefile*

Nous avons utilisé l'outil d'aide à la compilation *make*, en réalisant un *Makefile*. Notre projet se compose de plusieurs fichiers .h (contenant des prototypes) et .c (comprenant les différentes fonctions). En plus des fichiers **good.[ch]** et **stockex.[ch]**,contenant les prototypes et fonctions relatives aux ressources et aux places d'échanges/transactions, nous avons ajouté les fichiers **queue.[ch]** qui contiennent les prototypes de création d'une file de priorité et les différentes fonctions pour la manipuler, et **utility.[ch]** qui contiennent des fonctions utilitaires simplifiant l'écriture de la boucle principale.De plus, nous avons ajouté **strategy.[ch]** qui contiennent bien évidemment toutes les différents stratégies que nous avons implémenté.
Le fichier **project- ?.c** contient la fonction main, qui elle même contient toute la boucle de jeu (? étant l'*Achievement* courant).

La **figure 1** illustre les relations entre tous nos fichiers.

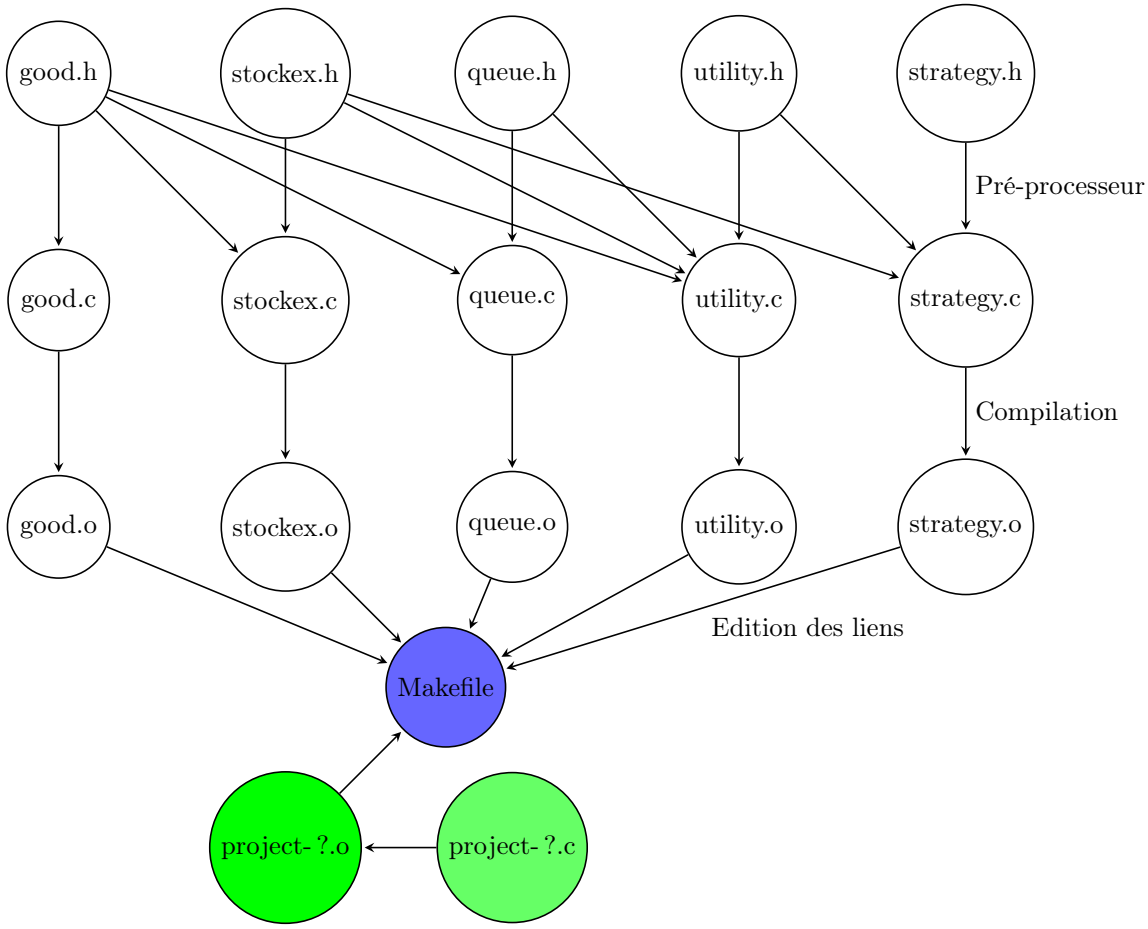

FIGURE 1 – Schéma de compilation

2 Structuration du projet

2.1 Structures initiales

Nous avons dû créer de nombreuses structures afin de mener à bien ce projet. Nous en détaillerons dans ce rapport les principales, afin de maximiser la clarté de la suite. Ces dernières sont disponibles dans des fichiers .h, **good.h** pour les structures en rapport avec les ressources, **stockex.h** pour les structures en rapport avec les places d'échange, **queue.h** pour celles en rapport avec les files de priorités de *teddies*.
Nous commencerons par détailler la version initiale de ces structures, c'est à dire celle de l'*Achievement* 0, puis nous montrerons les ajouts potentiellement apportés pour les *Achievements* suivants.

2.1.1 *Good*

— Enum *good* :

Afin de lister les ressources échangeables par les ours, nous avons utilisé une énumération, dans laquelle sont d'abord nommées les ressources, suivies d'une valeur nommée MAX_GOOD, elle même suivie d'une valeur $ERROR\_GOOD$ valant toujours -1. Cela permet, quelques soient les ressources, de garder la valeur MAX_GOOD égale au nombre de ressources listées.

— Structure *wallet*, un portefeuille :

Composée d'un tableau d'entiers, dont la taille est égale à MAX_GOOD (défini ci-dessus). Chaque case du tableau correspond à la quantité de la ressource (correspondant à l'indice de la case) présente dans ce portefeuille. La première case du tableau est nécessairement la quantité de miel, et chaque autre ressource possède un "prix" équivalent en miel, si bien qu'il est possible de connaître la valeur totale, en miel, d'un portefeuille.

2.1.2 *Stockex*

— Structure *transac*, une transaction :

Une transaction est simplement représentée par deux structures *wallets*, un portefeuille "*bought*", qui représente ce que le *teddy* va acheter, et un portefeuille "*sold*", qui lui représente ce que le *teddy* devra donner en échange.

— Structure *stockex*, la place d'échange :

Nous avons représenté une place d'échange par son nom, le nombre de transactions réalisables dans cette place ainsi qu'un tableau de transactions, donnant les transactions réalisables par les *teddies* sur cette place. La taille de ce tableau est, dans la structure, fixée et égale a $MAX\_TRANSAC$, c'est à dire le nombre maximale de transactions possible dans une place d'échange, d'où la nécessité de connaître le nombre de transactions réellement réalisable sut la place.

2.1.3 *Queue*

— Structure *teddy*

Un *teddy* est représenté dans le jeu, pour le différencier des autres, par un numéro, un entier noté "num". De plus, on doit savoir les ressources que possède le *teddy* ainsi que son temps passé à jouer, pour déterminer sa place dans la file de priorité. La structure *teddy* possède donc aussi un portefeuille "w" ainsi qu'un entier "$time\_played$". Nous avons de plus, pour la lisibilité et faciliter la détermination du grand gagnant au final, rajouté un entier "$value$" donnant la valeur totale du portefeuille en miel, c'est en quelque sorte le montant de son compte en banque. Nous sommes cependant conscient que cet entier n'est pas réellement une nécessité, et que si le calcul de celui-ci devait trop ralentir notre programme nous pourrions être amené à nous en passer. (Voir figure 3)

— Structure *queue*, la file de priorité :

L'élaboration de cette structure était l'une des principales problématiques de l'*Achievement* 0. Plusieurs choix s'offraient à nous pour implémenter une file de priorité. La première à laquelle nous avons naturellement pensé était d'avoir simplement un tableau de *teddies*, ordonnés suivant leur ordre de passage. Néanmoins, nous nous sommes vite aperçu que cette solution, bien que fonctionnelle, n'était pas optimale. En effet, avec cette représentation, nous devions à chaque tour modifier puis stocker les *teddies* ayant joués puis les recopier dans un tableau de file de priorité au tour suivant, ce qui n'était pas optimal en terme de mémoire et de temps de calcul.

Nous nous sommes alors dirigé vers une autre option, qui règle ce problème : implémenter une file de priorité comme un tableau de pointeurs, "tab_queue", qui pointent vers des *teddies* créés en début de partie. Ce tableau a, dans la structure, une taille fixe MAX_TED, c'est à dire le nombre maximal de *teddies* pouvant jouer une partie (que nous avons arbitrairement fixé à 20). Nous devions donc ajouter à la structure un entier "n", nécessairement inférieur à MAX_TED, représentant le nombre de *teddies* réellement présents dans la file de priorité. Afin de garantir que la file soit bien triée, nous ne la remplissons qu'à l'aide d'une fonction $queue\_\_push$, qui ajoute un *teddy* exactement à la place qui lui est réservée. Initialement, nous avions fais en sorte, arbitrairement et sans réelle réflexion, que le tableau soit trié dans des priorités (le *teddy* avec la plus grande priorité étant donc à l'indice 0), c'est à dire dans l'ordre croissant des temps de jeu. Néanmoins, cette implémentation impliquait que la fonction $queue\_\_pop$, qui fait sortir le *teddy* prioritaire de la file et qui est donc l'opération la plus élémentaire de manipulation du type abstrait *queue*, avait une complexité temporelle linéaire ($O(n)$). En effet, enlever le *teddy* d'indice 0 nécessitait alors de décaler tous les indices des $n-1$ *teddies* encore dans la file. Nous avons donc fait le choix de trier le tableau dans l'ordre inverse des priorités (ordre décroissant de temps de jeu). Ainsi, la fonction $queue\_pop$ se fait en temps constant, car les indices des *teddies* ne changent pas. Dans ces deux cas, cela n'influe pas sur la complexité de la fonction $queue\_push$, qui dépend de la priorité du *teddy* à ajouter dans la queue et est au minimum en $O(1)$, au maximum linéaire (nous la détaillerons plus tard dans ce rapport).

Une troisième option aurai été possible, celle de créer une liste chaînée de *teddies*. Cette solution permet

```
struct transac {
  struct wallet bought;
  struct wallet sold;
  const struct stockex* stock;       //la place d'échange où se déroule la transaction
  const struct stockex*  next_stock; //la place d'échange que débloque cette transaction
};
```

FIGURE 2 – Structure transac

d'avoir les même complexités que celle que nous avons choisi pour les fonctions *queue_pop* et *queue_push*. Cette option n'apporte donc pas de plus-value par rapport à notre solution choisie précédemment, nous avons donc arbitrairement conservé cette dernière. (Voir figure 4)

2.2 Mises a niveau des structures

Au file des *Achievements*, nous avons été amenés à apporter des modifications à certaines de ces structures afin de satisfaire à de nouveaux besoins. Nous détaillerons donc dans cette section ces quelques changements, *Achievement* par *Achievement*.

2.2.1 Achievement 1

Pour rappel, l'objectif de cet *Achievement* est d'implémenter plusieurs places d'échanges, déblocables par les *teddies* en fonctions des transactions qu'ils auront effectués précédemment dans la partie.
Notons tout d'abord que nous avons dûs créer plusieurs places d'échanges, que nous avons codées dans le fichier **stockex.c**. Nous avons ensuite, pour faciliter leur accès, créé un tableau de pointeurs pointant vers ces places d'échanges, que nous avons appelé *stocks*. La taille de ce tableau est égale à $MAX\_STOCKEX$, le nombre maximal de places d'échange possible. Voici donc maintenant les quelques modifications de structure apportées par l'*Achievement* 1 :

— Structure *transac* :

Afin de satisfaire les nouvelles exigences, chaque transaction doit posséder, en plus des ressources échangées, deux informations supplémentaires : tout d'abord la place d'échange dans laquelle se déroule la transaction, et ensuite quelle nouvelle place d'échange est débloquée par la transaction. Nous avons donc ajouté à la structure *transac* deux nouveaux champs, un champs $*stock$ qui pointe vers la place d'échange actuelle, et un champs $*next\_stock$ qui pointe vers le stock déblocable. (Voir figure 2)

— Structure *teddy* :

Nous avons modifié la structure *teddy* dans le même esprit. Nous y avons ajouté tout d'abord un champs $*current\_stock$, qui nous servira dans la boucle de jeu à entrer en mémoire du *teddy*, lorsqu'il a joué,

la place d'échange qu'il a débloqué (qui sera donc la place "actuelle" en début de tour). Ensuite, afin de connaître les places d'échanges déjà visitées, chaque *teddy* possède un tableau *tab_stock_name*, de taille *MAX_STOCKEX*, dans lequel est renseigné chacun des noms de ces places d'échanges. Enfin, afin de connaître la taille "utile" de ce tableau, un champs *nbre_stockex* renseigne le nombre de places d'échanges visitées par le *teddy*. Notre première idée pour que le *teddy* garde en mémoire les places d'échanges visitées était qu'il garde en mémoire l'indice de ces places d'échanges, déjà indexées grâce au tableau *stocks* défini précédemment, dans un tableau. Néanmoins, un problème de compilation, notamment avec l'option *valgrind*, que nous n'avons malheureusement pas réussi à comprendre, nous a poussé à changer de stratégie. Nous avons donc choisi assez arbitrairement l'implémentation d'un tableau de noms de places d'échanges. Une solution plus standard, que nous n'avons pas eu le temps de mettre en œuvre, aurai été plus simplement d'implémenter un tableau de pointeurs pointant vers les places d'échanges visitées. Néanmoins, la solution que nous utilisons est tout à fait fonctionnelle tant que les noms des places sont uniques. (Voir figure 3)

2.2.2 Achievement 2

L'objectif de cet *Achievement* est d'implémenter des stratégies positionnelles que les *teddies* peuvent utiliser afin d'augment de façon conséquente leurs chances de gagner, notamment face à des *teddies* jouant aléatoirement. Nous avons implémenter quatre différentes stratégies, que nous détaillerons plus tard. Nous noterons simplement ici que nous avons modifié la structure *teddy* ; en effet, nous avons du ajouter deux champs à cette structure. Premièrement, un entier "*strategy*" indiquant quelle stratégie le *teddy* utilise (une valeur 0 indiquant qu'il joue aléatoirement). Ensuite, pour les stratégies 3 ,4 et 5, nous avions besoin d'ajouter un pointeur vers ce qu'on appellera sa "transaction favorite".

```
1  struct teddy {
2    int num;
3    struct wallet w;
4    int time_played;
5    int value;
6    const struct stockex* current_stock;          //la place d'échange courante du teddy
7    int nbre_stockex;                             //le nombre de places où le teddy a déjà joué
8   char tab_stockex_names[MAX_STOCKEX][MAX_STR]; //tableau des noms de places où le teddy a déjà joué
9    int strategy;                                //le numéro de la stratégie
10   const struct transac* favourite_transac; // Sa transaction favorite
11 };
```

FIGURE 3 – Structure Teddy

2.2.3 Achievement 3

Ce dernier *Achievement* introduit la notion d'offre et de demande dans le jeu. Ainsi, La valeur de chaque ressource change tout au long du jeu, en fonction de sa disponibilité dans les places d'échanges. Afin de pouvoir accéder à tout moment à la quantité d'une ressource détenue par les *teddies* en tant constant (c'est à dire sans avoir à parcourir tous les *teddies* à chaque fois), nous avons fait le choix d'ajouter à la structure *queue* un portefeuille appelé *wallet_all_teddies*, qui contient pour chaque ressource sa quantité détenue par les *teddies*.

```
1 struct queue {
2   int n;          // Longueur de la file
3   struct teddy* tab_queue[MAX_TED]; // Tableau des ours présents dans la file
4   struct wallet wallet_all_teddies;// Portefeuille de tous les ours présents dans la file
5 };
```

FIGURE 4 – Structure Queue

3 Fonctions créées

Maintenant que nous avons définis les structures en place, nous pouvons discuter des principales fonctions qui nous sont utiles dans l'objectif d'implémenter une boucle de jeu. Ces dernières sont codées dans des fichiers .c, nommés **good.c**, **stockex.c**, **queue.c utility.c** et **strategy.c** . Certaines sont codées dans le fichier **project- ?.c** (? dépendant de l'*Achievement* en cours), qui contient la boucle principale de jeu, c'est à dire la fonction *main*, notamment la fonction récupérant les options de jeux tapées par l'utilisateur. Nous détaillerons le fonctionnement de ces fonctions ainsi que leurs complexités.

3.1 Fonctions de base

Commençons par détailler les principales fonctions créées dès l'*Achievement* 0 et servant de base au projet, étant globalement utilisées dans tous les *Achievements* suivants. Nous ne nommerons que les fonctions ayant une importance particulière de part leur apport au projet ou de part leur complexité temporelle.

3.1.1 *Good*

— Fonction *wallet__value* :

Cette fonction, qui calcule très simplement la valeur totale d'un portefeuille en parcourant celui-ci, a une complexité temporelle en $O(MAX\_GOOD)$, MAX_GOOD étant le nombre maximal de ressources échangeables possibles dans le jeu.

3.1.2 *Queue*

— Procédure *queue__push* :

Cette fonction est primordiale dans l'implémentation du jeu. Elle permet d'ajouter un *teddy* dans une file de priorité à la bonne place, dans le cas où cela est possible, c'est à dire que la file n'est pas pleine (cas qui ne devrai jamais arriver dans l'état actuel de notre jeu, étant impossible de lancer une partie avec un nombre de *teddies* supérieur à MAX_TED et une file étant justement pleine si le nombre de *teddies* dépasse MAX_TED).

Tout d'abord, nous n'oublions pas d'indiquer que la taille de la file a augmenté. La fonction parcoure la file dans le sens décroissant des indices, c'est à dire qu'elle part du *teddy* prioritaire, tant que les *teddies* qui y sont ont un temps de jeu inférieur au *teddy* que l'on souhaite inséré (ce qui détermine leur priorité) ou qu'il n'y a plus de *teddies* dans la file, afin de déterminé la place de celui-ci. Quand le *teddy* à placer se compare à un autre *teddy*, nous incrémentons l'indice de l'adresse de ce dernier dans la file, de sorte qu'il se décale d'une case vers le *teddy* prioritaire. Cette case est nécessairement libre, car soit elle n'a jamais été prise (c'est le cas si le *teddy* à décaler est le *teddy* prioritaire, sa place existe car le tableau n'est pas plein et nous avons au préalable incrémenter le nombre de *teddys* dans la file), soit le *teddy* précédemment à cette place a été décalé au tour précédent. A la fin de la boucle, il ne reste donc plus qu'à insérer le *teddy* à placer à la place restée libre. Dans le cas d'une file contenant N *teddies*, cette fonction effectue au maximum $O(N)$ opérations, dans le cas où le *teddy* à placer est le *teddy* le moins prioritaire, et au minimum $O(1)$ opérations, dans le cas où il est le *teddy* prioritaire (la boucle tant-que n'effectue même aucun tour dans ce dernier cas). C'est pour cette raison que nous avons parcouru les *teddies* du plus au moins prioritaire. En effet, les parcourir dans l'autre sens aurai nécessité, afin d'incrémenter les indices des *teddies* ayant un temps de jeu moins élevé, de parcourir dans tous les cas tout le tableau, c'est à dire que la fonction aurait toujours effectuées $O(N)$ opérations. De plus, afin de minimiser ce nombre d'opérations, si le *teddy* à placer possède le même temps de jeu qu'un autre *teddy* de la file, alors il sera placé devant lui. Cela sera d'autant efficace si tous les *teddies* ont le même temps de jeu, le *teddy* à placer étant donc le *teddy* prioritaire (utile pour la création initiale de la file en début de partie).

Pour finir, notons que pour créer une file de priorité, nous construisons tout d'abord une file vide à l'aide d'une fonction *queue__new*, puis nous ajoutons un à un les *teddies* dans cette file avec la fonction *queue__push*. Ainsi, une queue sera par construction toujours bien triée, et cela assure la correction de notre algorithme. Sa terminaison se démontre de façon triviale.

La figure 5 clarifie le fonctionnement de cette procédure.

— Fonction *queue__pop* :

Cette fonction permet de faire sortir le *teddy* prioritaire d'une file de priorité. Elle renvoie $NULL$ si on tente d'enlever un *teddy* à une queue vide. L'implémentation de notre file de priorité permet de réaliser cette opération très facilement. Le *teddy* prioritaire étant le dernier *teddy* de la file, il suffit de décrémenter le nombre de *teddies* présents dans la queue et de renvoyer ce *teddy*. Cette fonctions nécessite donc $O(1)$ opérations.

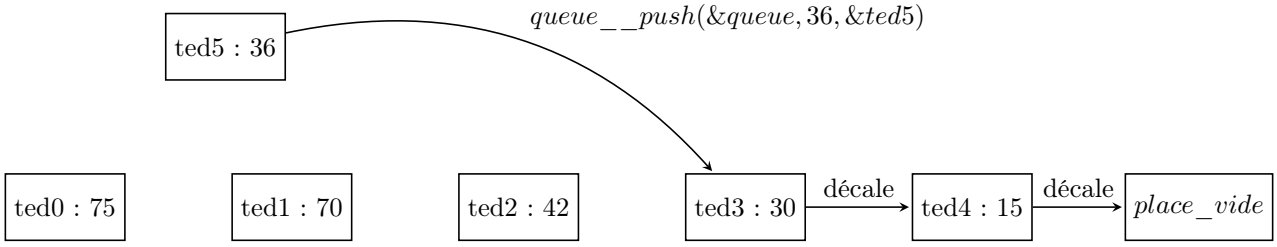

FIGURE 5 – Schéma de l'insertion d'un élément 'ted5', de temps de jeu '36', dans une file de priorité 'queue'
Ted? : t signifie que le teddy numéro '?' a un temps de jeu 't'

3.1.3 *project- ?*

— Procédure *parse_opts* :

Cette procédure spéciale permet à l'utilisateur du jeu de, s'il le souhaite, paramétrer certaines options (le nombre de joueurs n, le temps de jeu m, et l'initialisation d'un générateur aléatoire s). Cette fonction utilise la bibliothèque <**getops.h**>, qui permet de lire les instructions données dans le terminal de façon précise. L'interaction avec un utilisateur humain étant risqué, cette fonction doit savoir réagir correctement en cas de non respect des règles. Si l'utilisateur utilise une mauvaise syntaxe, l'ordinateur lui redemandera de rentrer ses paramètres, lui rappelant la syntaxe correcte. Si les valeurs entrées sont interdites (par exemple un nombre de joueurs négatif ou supérieur au maximum autorisé MAX_TED) ou qu'aucune n'est renseignée, des valeurs par défaut pour ces paramètres seront utilisées pour le jeu, et l'utilisateur en sera informé.

3.2 Fonctions ajoutées

Aux fonctions de base s'ajoutent des fonctions propres à chaque *Achievement*. Nous allons donc en présenter les principales, *Achievement* par *Achievement*

3.2.1 Achievement 1

Nous allons maintenant détailler les ajouts apportés par l'*Achievement* 1, dont le but, comme nous l'avons déjà évoqué précédemment, est d'implémenter dans notre jeu plusieurs places d'échanges, déblocables lors du jeu en fonctions des transactions effectuées par les *teddies*. Ces derniers devront, de plus, explorer le plus de places d'échanges possible, les choix de transactions ne seront donc plus totalement aléatoires. Nous avons, pour cela, dû implémenter quelques nouvelles fonctions qui s'ajoutent aux fonctions de base. Nous détaillerons deux d'entre elles, toutes deux présentent dans le fichier **utility.c**, car ce sont des fonctions utilitaires simplifiant l'écriture de la boucle principale.

— Fonction *utility__stockex_already_visited* :

Cette fonction prend en arguments une place d'échange et un *teddy*, et renvoie 1 si le *teddy* est déjà allé dans cette place d'échange, 0 sinon. Pour cela, la fonction parcourt le tableau des places déjà visitées par le *teddy* et compare les noms de ces places avec le nom de la place en argument. Cette fonction effectue donc

$O(nbr\_stockex)$ opérations au maximum, c'est à dire dans le cas où le *teddy* n'a pas déjà visité la place, en considérant la comparaison de chaînes de caractères comme une opération ($nbr\_stockex$ étant le nombre de places d'échanges visitées par le *teddy*).

— Fonction $utility\_\_transac\_choice$:

Cette fonction est à la base de l'*Achievement* 1. Elle permet à un *teddy* de choisir une transaction lors de son tour, en fonction de la place d'échange où il se trouve. Nous appellerons dans la suite "transaction intéressantes" une transaction permettant au *teddy* de débloquer une nouvelle place d'échange.

Tout d'abord, en utilisant la fonction $utility\_\_stockex\_already\_visited$ présentée précédemment, il ajoute au besoin la place d'échange à son tableau de places visitées. Ensuite vient enfin l'heure du choix. Il va parcourir les transactions disponibles dans la place d'échange, et vérifié pour chacune si la place d'échange déblocable a déjà été visitée. Si une transaction est intéressante pour lui, l'indice de cette transaction est stockée dans un tableau. Enfin, deux cas de figure se présentent : si aucune transaction n'était intéressante (on a ce constat grâce à un compteur de transactions intéressante) il tire au hasard une transaction parmi toutes celles de la place d'échange, et sinon il tire au hasard une case du tableau créé précédemment, ce qui reviens à tirer une transaction intéressante. La fonction renvoie alors l'adresse de la transaction choisie. Dans le pire des cas , cet algorithme a une complexité en $O(MAX\_TRANSAC * nbre\_stockex)$.

La figure 6 illustre l'utilisation de la fonction $utility\_\_transac\_choice$. On y suit deux *teddies*, rouge et vert, pendant le début du jeu.

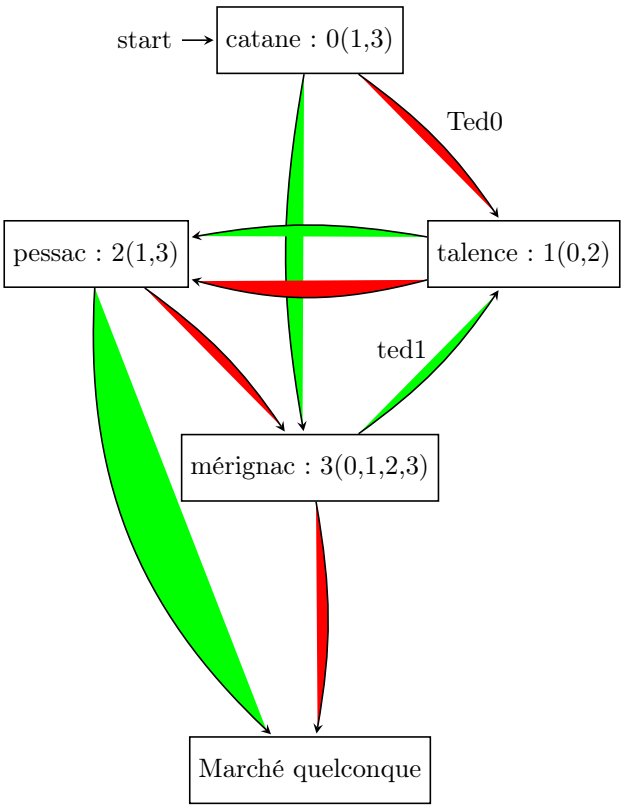

FIGURE 6 – Schéma illustrant des choix de transactions "intéressantes"

catane : 0(1,3) signifie que l'indice de "catane" est 0, et qu'on peut y faire 2 transactions qui débloquent respectivement les places d'indice 1 et 3

3.2.2 Achievement 2

L'*Achievement* 2 introduisant la notion de stratégie, nous avons du implémenter une fonction différente par stratégie, permettant de choisir stratégiquement une transaction en fonction de la place d'échange courante. La fonction *utility__transac_choice* introduite dans l'*Achievement* 1 servira de stratégie de base, la stratégie 0, sous le nouveau nom de *strategy__random_choice*. Notons tout d'abord que toutes ces fonctions sont implémentées dans le fichier **strategy.c**. Ensuite, notons que *teddies* ne pourrons effectuer que une transaction par tour, ce qui limite mais simplifie les stratégies possibles.

— Stratégie 1 :

Cette première stratégie est implémentée dans la fonction *strategy__strategy*1. Son fonctionnement est extrêmement basique. Le *teddy* stratège va, dans la place d'échange, choisir la transaction ayant le plus grand bénéfice brut (c'est à dire en valeur de miel) qu'il peut réaliser. Il a donc effectivement la meilleure transaction dans cette place d'échange en terme de bénéfice. Néanmoins, on aperçois facilement les limites de cette stratégie, qui ne se base que sur la transaction présente et ne fais aucun plan sur du "long terme". Comme nous pouvons le voir sur la figure 7, les transactions choisies par le *teddy* étant en vert, un problème possible est que le *teddy* reste bloqué entre quelques places d'échanges dont les profits possibles ne sont pas très élevés, alors que se rendre sur d'autres places pourrait être beaucoup plus rentable. Notons tout de même

que cette stratégie fonctionne globalement plutôt bien contre des *teddies* jouant avec la stratégie 0.

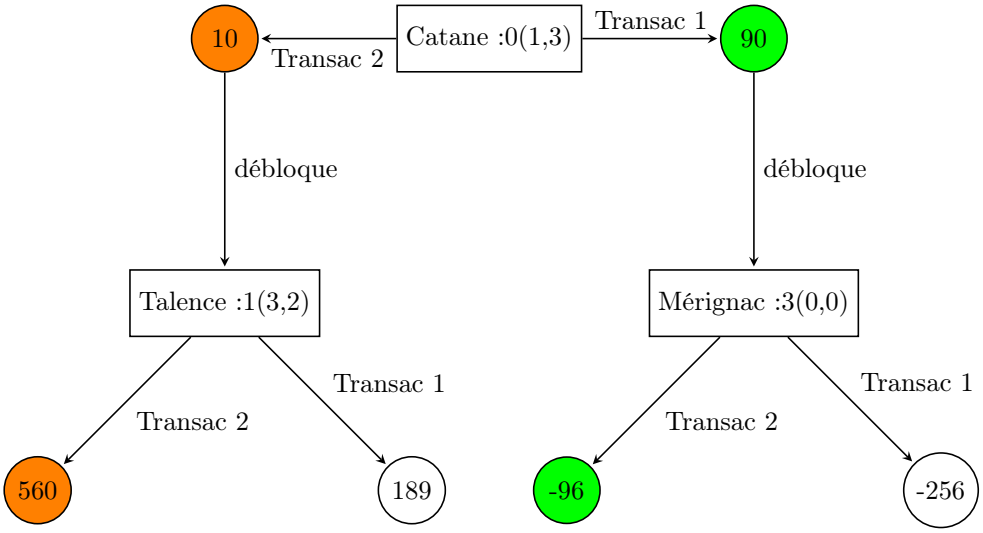

FIGURE 7 – Schéma illustrant un défaut de la stratégie 1

— *Catane : 0(1,3) signifie que l'indice de "catane" est 0, et qu'on peut y faire 2 transactions qui débloquent respectivement les places d'indice 1 et 3.*

— *La valeur dans les cercle représente le bénéfice de la transaction correspondante.*

— Stratégie 2 :

La deuxième stratégie est très proche de la première. Afin de remédier un minimum au problème que nous avons expliciter, le *teddy* utilisant la stratégie 2 va calculer pour chaque transaction disponible dans la place actuelle la somme du bénéfice de cette transaction avec le bénéfice maximal des transactions disponibles dans la place d'échange débloquée par la première transaction. De cette manière, il évitera de tomber directement dans certains pièges comme le ferai un *teddy* utilisant la stratégie 1 (dans la figure 7, il choisira le chemin orange, bien plus profitable que le vert). Néanmoins, il pourrait de la même façon se trouver bloqué dans une boucle de places d'échanges rapportant moins de miel que d'autres. On pourrai alors, par récurrence, effectuer k fois le principe de cette stratégie (il visite "à l'avance" k places d'échanges et fait la somme de toutes les transactions pour chaque chemin). Néanmoins, la complexité de cet algorithme serai très grande (au maximum $O(MAX\_TRANSAC^k)$ si chaque place possède $MAX\_TRANSAC$ transactions), et possède un très gros défaut : si le *teddy* ne possède plus assez de ressources pour réaliser les transactions choisis à l'avance au moment où il arrive sur la place d'échange, la stratégie aura été veine.

— Stratégie 3 :

Pour cette stratégie, nous avons besoin que l'ours puisse "tricher" par rapport aux règles de base du jeu. En effet, nous considérons qu'il peut ne pas changer de place d'échanges s'il le souhaite. Il mémorise, grâce à la modification de la structure *teddy* que nous avons effectué, la transaction la plus profitable qu'il ait rencontré lors de ses tours. Lorsqu'il rencontre cette transaction, il l'effectue en boucle au file de ses tours jusqu'à ne plus avoir les ressources nécessaires pour la faire. Il choisi alors la transaction que donnerait la stratégie 1. S'il rencontre alors une stratégie plus profitable, il met à jour sa transaction favorite, et effectue le même

stratagème. A chaque tour, il effectue donc soit sa stratégie favorite s'il le peut, soit il applique la stratégie 1 et il change de place d'échanges.

— Stratégie 4 :

La stratégie 4 est quasiment identique à la stratégie 3, à la différence près que le *teddy* utilise la stratégie 2 à la place de la stratégie 1. Il peut alors apercevoir une transaction qui serai plus profitable que sa transaction favorite et donc aller vers celle-ci.

— Stratégie 5 :

Dans cette dernière stratégie, le *teddy* tente de visiter toutes les places d'échange afin de déterminer la véritable transaction la plus rentable, puis il agit avec la stratégie 4 en ayant cette transaction comme transaction favorite.

3.2.3 Achievement 3

Dans ce dernier *Achievement*, les valeurs des ressources changent en fonction de leur disponibilité dans les places d'échange. Leur valeurs augmentent si elles sont rares, et diminuent sinon. Plus précisément, si la transaction qu'a choisit un *teddy* implique la vente d'une ressource 'g' alors qu'elle en excès dans les marchés (c'est à dire que l'ensemble des ours possèdent moins de 10% de l'ensemble de 'g' disponible dans le jeu), la valeur de 'g' diminue de 1. De la même manière, si cette transaction implique l'achat d'une ressource 'h' rare (l'ensemble des ours possédant déjà plus de 90% du 'h' disponible), la valeur de 'h' augmente de 1.

— Modification de la fonction $good\_\_value$: Afin de pouvoir modifier les valeurs des ressources en cours de partie, nous avons du modifier la fonction $good\_\_value$ qui auparavant renvoyai, avec une ressource énoncée en paramètre, la valeur de celle-ci à l'aide d'un *switch* dont les valeurs de chaque ressource étaient codé en dur dans la fonction. Nous avons donc créé un tableau statique d'entiers, représentant les valeurs de chaque ressources et pouvant être modifié au cours de la partie. $good\_\_value$ n'a alors plus qu'à renvoyer la valeur dans le tableau correspondant dont l'indice correspond avec la ressource.

— Fonctions $utility\_\_good\_greater\_limit$ et $utility\_\_good\_lesser\_limit$:

Ces deux fonctions renvoient respectivement si une ressource est en possession des *teddies* à plus de 90% ou à moins de 10%. Grâce au changement que nous avons réalisé sur la structure *queue* (l'ajout d'un portefeuille donnant le nombre totale de chaque ressources possédées par les *teddies*), nous pouvons réaliser cette opération en temps constant. Notons que les quantités de ressources en jeu sont renseignées dans un tableau statique d'entiers et que chacune de ces valeurs peut être retourné par la fonction nommée $good\_\_total\_amount$, prenant en argument une ressource.

— Fonction $utility\_\_check\_total$:

Cette fonction vérifie qu'une transaction est réalisable sans que les *teddies* ne possède plus de 100% des ressources disponibles. En d'autres termes, elle parcourt les MAX_GOOD cases du portefeuille de la file de priorité et les compare avec les quantités maximales en jeu de chaque ressources. Cette fonction effectue donc clairement $O(MAX\_GOOD)$ opérations.

— Fonction $utility\_\_good\_set\_value$ Cette fonction permet de modifier au besoin la valeur d'une ressource

avant d'effectuer une transaction. Elle parcourt les portefeuilles caractérisant la transaction et détermine si les valeurs doivent être modifiés avec les fonctions $utility\_\_good\_greater\_limit$ et $utility\_\_good\_lesser\_limit$, qui s'exécutent en temps constant. Cette fonction réalise donc $O(MAX\_GOOD)$ opérations.

4 Boucle principale

Nous possédons maintenant tous les outils nécessaires à l'implémentation de la boucle de jeu principale dans le *main*. Nous allons en détailler les principales étapes. Les modification du main sont assez minimes entre les *Achievement*, nous détaillerons donc principalement la boucle principale de la version de base du jeu, puis nous reviendrons sur les quelques changements apportés par les *Achievements* suivants.

4.1 Version de base

1. Tout d'abord, après avoir initialisé n, m et s (respectivement le nombre de joueurs, le temps de jeu total et le générateur aléatoire), nous créons un tableau de n *teddies*, ayant tous le même porte-monnaie et un temps de jeu logiquement nul. Cela nécessite $O(n)$ opérations. Nous créons alors une file de priorité initiale, en ajoutant avec le fonction *queue_push* les adresses de ces *teddies* dans une queue initialement vide. Cette partie du programme exécute la fonction $queue\_\_push$ n fois à des files de plus en plus grandes (de 0 à n-1), mais l'implémentation de la fonction permet, comme tous les *teddies* ont le même temps de jeu, de n'effectuer que $O(1)$ opération par appel de $queue\_\_push$. On peut donc considérer qu'elle exécute $O(n)$ opérations.
2. L'initialisation effectuée, le jeu peut alors véritablement commencer. On effectuera une boucle tant que le temps de jeu globale sera inférieur à m, le temps à jouer. Dans le pire des cas, si chaque tour ajoute un temps 1 au temps de jeu, m tours seront réalisés. A chaque tour de boucle, le *teddy* prioritaire est retiré de la liste à l'aide de $queue\_\_pop$ (de complexité $O(1)$). Chaque transaction de la place d'échange étant rangé dans un tableau, un nombre tiré aléatoirement permet de renvoyer la transaction aléatoire dont l'indice correspond.
3. Cette transaction choisie, on détermine combien de fois le *teddy* l'effectuera de la façon suivante : on tire un nombre N aléatoirement, inférieur à une valeur $MAX\_OPERATIONS$, le maximum de transactions autorisé par tour. Si le Teddy est en capacité d'effectuer N fois la transaction, il le fait. Sinon, il tire à nouveau un nombre aléatoire N' qui doit être inférieure strictement à N, et recommence tant qu'il ne peux pas effectuer de transactions. Sachant qu'il pourra forcement réaliser 0 transactions (les dettes étant interdites), la terminaison de cet algorithme est assurée. Dans le pire des cas, le *teddy* réalisera cette opération $MAX\_OPERATIONS$ fois, si il ne possède pas assez pour faire la transaction et que N décrémente seulement de 1 à chaque fois.
4. La transaction effectuée, le temps de jeu du *teddy* est augmenté en conséquence (de 1 s'il n'a pas réalisé de transaction, de $1 + log(N)/log(2)$ sinon) , tout comme le temps global de la partie. La valeur de son portefeuille est recalculée avec la fonction $wallet\_\_value$ (avec une complexité en $O(MAX\_GOOD)$), puis le *teddy* est replacé dans la file de priorité à l'aide d'un $queue\_\_push$ (de complexité maximale $O(n)$). Ceci

marque la fin du tour du *teddy*. Un nouveau tour débute alors, si le temps imparti n'est pas dépassé.

5. Pour finir le jeu se termine alors en affichant à l'écran le classement des *teddies* et notamment le grand gagnant ainsi que la valeur de son incroyable fortune. Cette opération se déroule en $O(n_2)$ car nous avons trié les *teddies* par tri bulle afin d'afficher le classement. Une optimisation serai de n'afficher que le *teddy* gagnant, ce qui permettrai de ne pas avoir à trier les *teddies* et de ne réaliser donc que $O(n)$ opérations pour simplement parcourir les *teddies* et comparer leurs valeurs de portefeuille.

La figure 8 illustre cette boucle de jeu.

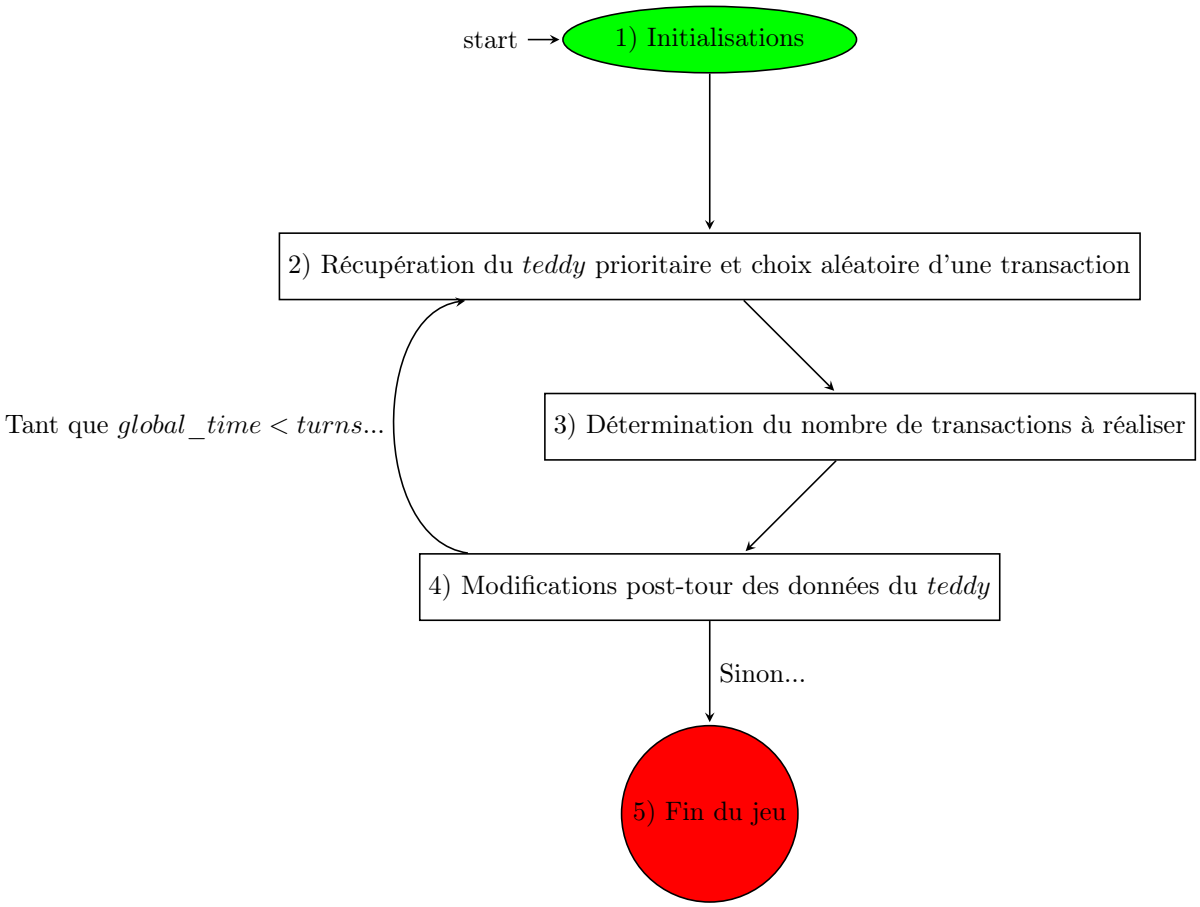

FIGURE 8 – Schéma de la boucle de jeu

4.2 Modifications

4.2.1 Achievement 1

La boucle principale est très peu changée ; en effet, seuls deux changements sont à noter. Premièrement, le choix précédemment aléatoire d'une transaction est maintenant remplacé par la fonction *utility__transac_choice*. Ensuite, après avoir joué, le *teddy* doit mettre à jour la place d'échange dans laquelle il devra jouer au tour prochain.

4.2.2 Achievement 2

A nouveau la boucle principale change très peu entre les *Achievements* 1 et 2. En effet, la grande majorité des changements se situent dans la fonction de choix de transaction. Nous avons seulement enlever la déterminations aléatoire du nombre de transactions à réaliser par le *teddy* durant son tour, celui-ci étant maintenant limité à 1 si le *teddy* peut jouer, 0 sinon. On peut d'ailleurs noté que le temps de jeu augmente toujours de 1 à chaque tour, et que le temps de jeu final est donc maintenant le nombre de tours de la boucle de jeu.

4.2.3 Achievement 3

Dans cet *Achievement* final, la condition pour qu'un *teddy* réalise la transaction qu'il choisi se durcit : il doit toujours d'une part posséder les ressources suffisantes, mais d'autre part les ressources qu'il veut récupérer doivent être disponible en quantité suffisante dans les places d'échanges, ce qu'on vérifie à l'aide de *utility__check_total*. Ensuite, avant que le *teddy* réalise l'échange s'il est possible, on modifie au besoin les valeurs des ressources à l'aide de *utility__good_set_value* Le reste de la boucle n'est pas modifié.

5 Tests effectués

Afin de s'assurer de la robustesse de nos structures ainsi que de la correction de nos algorithmes, nous avons tout au long des différents *Achievement* élaborer des tests. Ceux-ci sont stockés dans un dossier *tst* à part du reste du programme, et sont compilés et exécutés à l'aide de la commande bash *make test*, définie dans le $Makefile$. Nous présenterons dans ce rapports les tests principaux effectués. Notons que tous les tests présentés donnent de bons résultats.

5.1 *Queue*

— Afin de tester la structure *queue* ainsi que les fonctions *queue__push* et *queue__pop*, on remplie une queue vide avec des *teddies* (non triés et codés directement en dur dans le code) à l'aide de *queue__push* et on vérifie que les *teddies* renvoyés par des appels à *queue__pop* renvoient bien le bon *teddy* à chaque fois.

— Ensuite, nous contrôlons la robustesse de notre structure *queue*. Premièrement, un test tente d'insérer, par la fonction *queue__push*, un *teddy* dans une queue pleine. On vérifie alors que ce *teddy* n'a pas été ajouté à la queue. Deuxièmement, nous tentons de retirer un *teddy* d'une queue vide à l'aide de la fonction *queue__pop*. Nous vérifions alors que le fonction retourne bien $NULL$.

5.2 Choix de transactions

Comme précédemment nous appellerons transaction intéressante une transaction permettant au *teddy* de visiter une nouvelle place d'échange. Nous avons initialisé différents *teddies* ayant visités des places d'échanges prédéfinis, et nous avons vérifié que ces *teddies* choisissent bien une transaction intéressante disponible dans la place d'échange courante.

5.3 Stratégies

Nous avons vérifié, sur des situations codées en dur, que les *teddies* utilisant nos stratégies avaient bien le comportement attendu.

5.4 Modifications des valeurs des ressources

Nous testons si les fonctions modifiants conditionnellement les valeurs de quelques ressources fonctionnent bien, à partir de situations prédéfinis. Nous testons notamment la réaction du programme si les ressources ne sont pas présentes en jeu (auquel cas il ne fait pas la transaction).

6 Pour finir

6.1 Programme et ses options

6.1.1 Code principal

En exécutant le programme à l'aide de la commande bash **./project**, $n = 2$ teddies se mettent à jouer pendant $m = MAX\_PARTY\_TIME = 1000$ tours. Le programme utilise par défaut le temps comme valeur de "seed s". A la fin de notre programme, le classement s'affiche, puis le *teddy* gagnant et enfin l'ensemble de places d'échange visités par ce *teddy*. En outre, l'utilisateur peut choisir les options p, m et s en utilisant la notation suivante dans le terminal : **./project -n 14 -m 500 -s 12**. Ces valeurs sont des exemples. Si cette syntaxe n'est pas respectée, un message d'aide apparaît et l'utilisateur doit à nouveau taper la commande. Si les valeurs rentrées ne sont pas compatibles, les valeurs par défaut sont utilisées.

6.1.2 Code de test

En lançant la commande *bash* **make test** dans le terminal, les codes de test sont directement compilés et exécutés. Il s'affiche alors à l'écran le nombre de tests réussis par catégorie de test, puis le nombre globale de tests réussis.

6.2 Difficultés rencontrées

D'une façon très générale ,ce premier projet nous a posé de nombreuses difficultés concernant la programmation. En effet, nous avons du rapidement être capable de bien maîtriser les commandes *bash*, en particulier les commandes *git* permettant de manipuler nos fichiers sur la forges, ainsi que la programmation C, que nous ne connaissions pas du tout deux mois avant le début du projet. Outre les innombrables erreurs de segmentations que nous avons eu du mal

à résoudre, nous avons mis du temps à comprendre l'intérêt d'un code clair et surtout homogène dans sa syntaxe. Cela nous a fait perdre énormément de temps, car nous avons été obligé à plusieurs moment de revoir tout notre code afin de changer le nom de certaines fonctions, harmoniser les espaces et tabulations, changer les commentaire, changer des fonctions de place... Un autre domaine que nous ne maîtrisions pas du tout non plus était l'élaboration d'un rapport. Outre les difficultés relative que nous avions à écrire en LaTeX, notre première version de ce rapport était très mal organisé, les phrases étaient mal formulées et quelques éléments essentiels étaient manquant quand certains inutiles étaient présents. Enfin, nous avons au début négligé l'intérêt, pourtant primordial, d'effectuer des tests complets.

Plus spécifiquement, nous avons tout d'abord eu quelques difficultés dans l'implémentation de la file de priorité, que nous avons changé à plusieurs reprises avant d'en arriver à notre version finale. Comme nous l'avons explicité dans le rapport, nous avons eu un problème pour faire garder en mémoire à un *teddy* les places d'échanges qu'il a visité. Nous avons réglé ce problème en contournant quelque peut celui-ci et en utilisant une solution assez spéciale (utiliser les noms de ces places d'échange), ce qui ne ne nous plaît évidemment pas mais qui fut une nécessité.

Dans le même registre, un $Warning$ qui apparaissait lors de la compilation de notre code et que nous n'arrivions pas à enlever nous avait pousser à demander, dans le $Makefile$, que le compilateur ignore ce $Warning$, ce qui était évidemment une très mauvaise solution. Nous avons finalement réussi à régler ce problème. Enfin, ayant commencé les *Achievement* 2 et 3 assez tardivement, le temps fut un problème vers la fin du projet, mais nous voulions absolument réussir à implémenter tous les *Achievements* disponibles. La dernière semaine fut donc très chargée, car cela s'additionnait avec de nombreux changements à faire sur notre rapport et un nettoyage complet du code à réaliser.

6.3 Gestion de temps

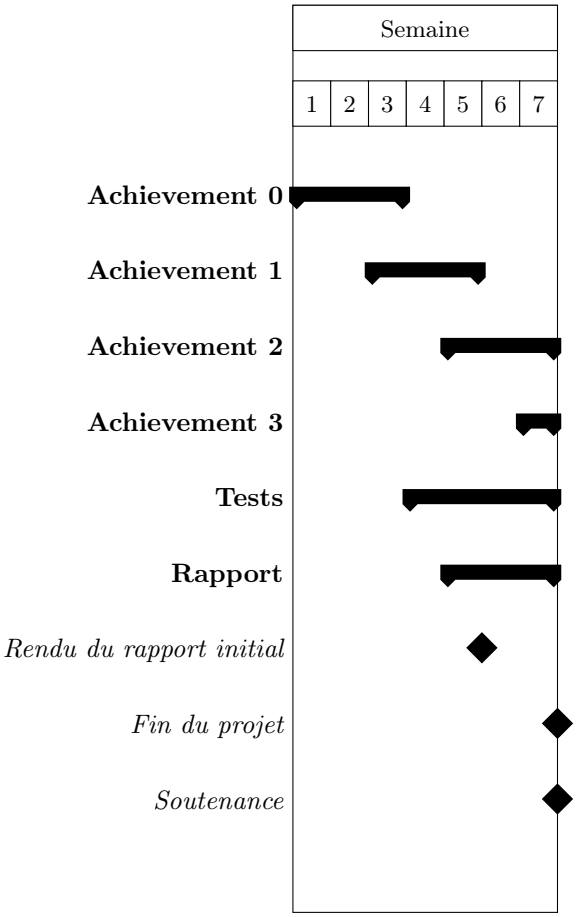

FIGURE 9 – Diagramme de Grantt du projet

7 Conclusion

En conclusion, ce premier projet nous a permis d'approfondir nos connaissances, que ce soit en programmation C, en environnement de travail avec l'utilisation des commandes Bash sous Linux, du LATEX, mais également en algorithmique avec les réflexions préliminaires à l'élaboration du code ainsi que les calculs de complexité. Nous avons également développé notre esprit d'équipe et de collaboration, ce qui est pour nous une préparation essentielle à ce qui nous attend dans les années à venir.

8 Remerciements

Nous remercions tout d'abord M. David RENAULT pour les remarques, aides et observations qui nous ont été données soit en cours, soit par courriel, ainsi que pour l'élaboration de ce projet original. De plus, nous remercions M. Michaël CLEMENT, qui nous a accompagné tout au long du projet en nous encadrant et en répondant à toutes nos questions. Ses réflexions sur l'importance de la rigueur dans la programmation nous ont été utile pour ce projet et nous seront probablement utiles pour le reste de notre vie de programmeur.

9 Références

9.1 Internes

- https://www.labri.fr/perso/renault/working/teaching/projets/
- https://www.labri.fr/perso/fmoranda/pg101/

9.2 Externes

- https://openclassrooms.com/
- https://stackoverflow.com/
- https://tex.stackexchange.com/
- https://www.xm1math.net/doculatex/
- http://www.texample.net/
- https://www.overleaf.com/