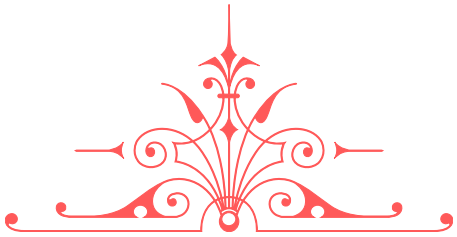
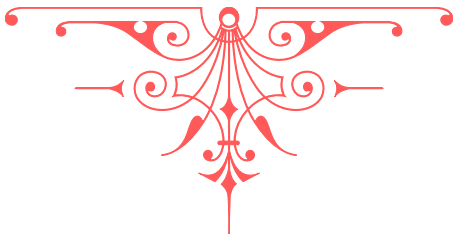


Rapport de projet S5 C : Atomic Teddy Investors

Par : Walid Laaziri & Hicham Zghari

Encadrants: M. Clément & Mme Desainte-Catherine



Bordeaux INP
ENSEIRB
MATMECA

Table des matières

1 Introduction

Ce rapport vise à présenter notre démarche de réalisation du projet de programmation de 1^{re} année. Le projet consiste en l'implémentation d'un jeu de stratégie où des ours, réalisent des échanges de ressources tour à tour, sur différentes places d'échanges, en fonction de leur capital.

1.1 Contexte

FIGURE 1 – Affiche du film *La Fameuse Invasion de la Sicile par les ours*, de Lorenzo Mattoti

Le contexte s'inspire du roman de Dino Buzzati, *La Fameuse Invasion de la Sicile par les ours*, qui a fait l'objet d'une adaptation cinématographique de Lorenzo Mattoti (affiche en FIGURE **??**). Après l'invasion de la Sicile par les ours, ces derniers s'intègrent dans les activités humaines. Ils sont attirés par l'économie de marché et décident donc de s'y consacrer corps et âme. Ainsi, ils rivalisent de stratégies afin d'amasser des fortunes colossales. Leur unique monnaie d'échange est le miel, qui leur est si cher.

C'est dans ce contexte que s'inscrit le jeu que nous allons implémenter au cours de ce projet. En effet, les ours, que l'on surnomme affectueusement `teddys`, jouent chacun leur tour sur les places d'échanges. Au cours d'un tour, un `teddy` effectue une transaction sur une place d'échanges. Cette transaction le redirige, ensuite, vers une nouvelle place d'échanges.

1.2 Description de la demande

Afin de modéliser ce jeu, les teddys sont placés initialement dans une file de priorité. L'implémentation de cette file de priorité est le coeur du jeu. De plus, il est nécessaire d'établir différentes stratégies pour les teddys et d'évaluer la stratégie optimale qui assure le gain. Enfin, on améliorera le jeu en ajoutant une limitation de ressources dans le monde des `teddys`. Ainsi, les valeurs des ressources pourront évoluer en fonction de leur disponibilité.

1.3 Problématique

Dans le cadre de la réalisation de ce jeu en C, plusieurs problématiques se profilent.

→ En pratique, quelle structure devrions-nous choisir pour implémenter la file de priorité ? En effet, il est primordial de faire un choix qui soit adapté aux opérations d'ajout et de retrait de teddys dans la file, avec des complexités optimales.

→ Comment garantir la robustesse du projet à différentes implémentations de la même interface ?

→ Quelles sont les stratégies positionnelles envisageables ? Serait-il possible de définir une fonction qui permette de déterminer la stratégie optimale ?

→ Comment limiter les ressources disponibles et faire évoluer leur valeur en fonction de leur disponibilité ?

2 Organisation du travail

2.1 Répartition des tâches

Le travail a été divisé de manière égale et complémentaire. Nous nous sommes répartis les différentes tâches de codage. Quant aux réflexions autour des problématiques algorithmiques et des choix d'implémentations, nous en débattions pendant les séances de projet. Cette organisation nous permettait d'avoir les idées claires afin d'entamer ou de continuer la production du code chez nous, en dehors des séances. Il a fallu faire preuve de coordination.

2.2 Outils utilisés

2.2.1 Git

Nous avons utilisé le système de gestion de version Git. Les différentes versions du code sont stockées sur la `Forge` de L'ENSEIRB-MATMECA (serveur `Thor Project Manager`). De plus, chacun a accès à un dépôt local dans son arborescence de fichiers. Cette nouvelle manière de travailler a été rude à mettre en place. Nous avons engendré plusieurs conflits au début de notre projet. Néanmoins, nous avons pu apprécier les avantages de ce système. En effet, il nous permet de visualiser l'évolution de notre projet et de corriger d'éventuels mauvais choix, grâce à la disponibilité de toutes les versions intermédiaires.

2.2.2 Emacs

Nous avons utilisé l'éditeur de texte Emacs que nous utilisons régulièrement dans les autres unités d'enseignement.

2.2.3 LateX

Nous avons rédigé ce rapport en LaTeX. Ce dernier est un langage que nous avons appris à utiliser en cours d' "Environnement de Travail" (IF104).

2.3 Arborescence des fichiers

Nous avons utilisé l'outil de compilation `make` en créant un fichier `Makefile`. Nous avons organisé nos données dans plusieurs fichiers `.c` et `.h`. Les fichiers `queue.h` et `good.h` contiennent toutes les structures et les signatures de fonctions utilisées respectivement dans les fichier `queue.c` et `good.c`. Les fichiers `stockex.h` et `transac.h` contiennent les structures et les signatures des fonctions utilisées dans les fichiers `stockex.c`. Le fichier `initialisation.h` contient les tructures et les signatures de fonctions utilisées dans le fichier `initialisation.c`. Le fichier `project-2.c`, quant à lui, contient la boucle principale du jeu et fait appel à toutes les fonctions et structures prédéfinies dans les fichiers `.h`. On peut schématiser les liens entre les différents fichiers par la FIGURE 2 ci-dessous.

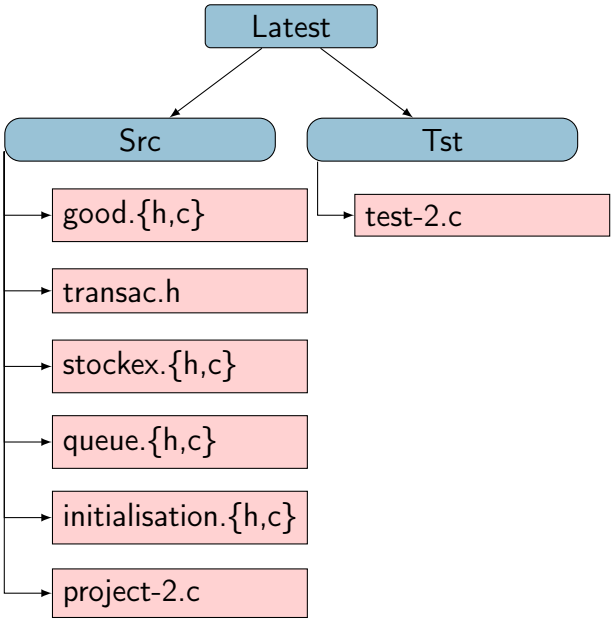


FIGURE 2 – Arborescence des fichiers

2.4 Structures de données

Les structures de données ont été un point essentiel de la réflexion sur le projet. En effet, nous les avons implémentées de manière à ce que notre code soit le plus compréhensible possible (noms de variables représentatifs). Nous avons rédigé l'ensemble de notre code en anglais (noms de variables, noms des fonctions, contantes et structures de données). Néanmoins, par soucis de compréhension et de respect de langue du sujet, nous avons choisi de retourner les

sorties printf en français.

2.4.1 Constantes

→ **Constantes globales**
Les constantes globales sont : `MAX_GOOD` qui représente le nombre maximal de ressources, `MAX_PLAYERS` qui représente le nombre maximal de joueurs et `MAX_STOCKEX` qui représente le nombre maximal de places d'échanges.

→ **Constantes définies dans `project2.c`**
Dans `project2.c`, on définit `seed` qui représente la graine aléatoire, `players` qui représente le nombre de joueurs, `turns` qui représent le nombre de tours, `max_party_time` qui représente le nombre maximal de parties et `max_transactions_number` qui représente le nombre maximal de transactions réalisables par un `teddy` pendant un tour.

2.4.2 Structures basiques

→ **Structures good et structure wallet**

Dans le fichier `good.h`, nous avons repris l'énumération `enum good` et la structure `struct wallet` proposées dans le sujet. L'énumération `good` sert à énumérer toutes les ressources qui existent dans le monde des `teddys`. Quant à la structure `struct wallet`, elle permet de représenter le porte-feuille d'un `teddy` et contient un tableau d'entiers non signés dont la taille est égale au maximum des ressources possibles. Chaque case du tableau représente la quantité disponible pour une ressource.

Nous avons implémenté des fonctions permettant de manipuler l'enumération `good`. La première fonction est `good__string`. Cette dernière utilise un `switch` qui permet de retourner la description d'une ressources entrée en paramètre. La deuxième fonction, nommée `good__value`, permet, quant à elle, de retourner la valeur en `HONEY` correspondant à la ressource entrée en paramètre de type `enum good`. De plus, nous avons ajouté deux fonctions afin de permettre la limitation des ressources (Achievement 3). La fonction `good__total_amount` retourn la quantité maximale pour une ressource et la fonction `good__set_value` qui permet de modifier la valeur d'une ressource. Ces deux dernières fonctions se font en temps constant.

→ **Structures stockex et transac**

La structure de données `struct stockex` (Code 2) représente une place d'échange. Elle comporte le nom de la place d'échange de type `char *`, le nombre de tansactions réalisable sur la place d'échange, de type `int` et un tableau de transactions de type `struct tansac` de taille constante `MAX_TRANSAC` (fixée à 20 dans le sujet). Cette structure `transac` (Code 1) représente une transaction. Une transaction étant un échange de porte-feuilles (achat et vente), la structure contient deux structures `wallet` et un

pointeur vers une autre place d'échange. En effet, chaque transaction doit mener à une autre place d'échange où le teddy pourra réaliser une autre transaction.

Code 1 – Structure `transac`

```
struct transac{
    struct wallet achat;
    struct wallet vente;
    const struct stockex* next_stockex;
};
```

Code 2 – Structure `stockex`

```
struct stockex{
    char* place;
    int nombre_transactions;
    struct transac transactions[Max_transac];
};
```

On utilise, de plus, plusieurs fonctions qui utilisent la `struct transac` et qui sont relatives aux transactions. Tout d'abord, la fonction `starting_stockex()` permet d'initialiser une place d'échange. Ensuite, la fonction `our__starting_stockex` qui prend un entier et retourne la place d'échanges qui a pour indice cet entier dans le tableau qui contient les différentes places d'échanges. La fonction `stockex__name`, quant à elle, permet de retourner le nom de la place d'échanges qui lui est passée en paramètre. La fonction `stockex__num_transactions` retourne le nombre de transactions disponibles dans une place d'échanges. Une autre fonction, nommée `stockex__transaction`, prend en paramètres une structure `stockex` et un entier, et retourne la transaction de la place d'échanges qui a pour indice l'entier entré en paramètre. Ensuite, la fonction `transac__in_wallet` qui retourne les rassources reçues par la places d'échanges au cours d'une transaction, entrée en paramètre. Réciproquement, la fonction `transac__out_wallet` retourne les ressources achetées par le `teddy` au cours d'une transaction entrée en paramètre. De plus, nous avons implémenté la fonction `transac__next_stockex` qui renvoie la place d'échange à laquelle mène la transaction passée en paramètre. Toutes les fonctions présentées précédemment sont de complexité en temps constante. Enfin, nous avons implémenté une fonction `transac__stockex` qui retourne la place d'échange qui contient la transaction qui est passée en paramètre. La complexité en temps de cette fonction est $O(nm)$ où n est le nombre moyen de transactions par places d'échanges.

→ Structure teddy

La structure `struct teddy` (Code 3) représente un joueur. Elle contient l'identifiant du joueur qui est de type entier, le porte-feuille du joueur qui est de type `struct wallet`, sa priorité (qui est égale à son temps de jeu dans le sujet) qui est de type entier, sa place d'échange actuelle qui est un pointeur vers une structure `stockex`. De plus, nous y avons inclus un tableau de pointeurs vers des structures `stockex` qui représente les places d'échanges visitées par le joueur. Enfin, elle contient également un entier qui représente le nombre de places d'échanges visitées et un entier qui ne peut prendre comme valeur

que 0 ou 1. Cet entier permet de savoir si un teddy utilise une stratégie au cours du jeu (l'entier vaut 1) ou s'il agit aléatoirement (l'entier vaut 0).

Code 3 – Structure `teddy`

```
struct teddy {
    int name;
    struct wallet w;
    int time;
    const struct stockex* current_stockex;
    const struct stockex* stockex_visited[MAX_STOCKEX];
    int number_stockex_visited;
    int optimal;
};
```

3 Implémentation de la file de priorité

3.1 Implémentation pratique

La file de priorité est l'élément central du jeu. Son implémentation nous a demandé plusieurs heures de réflexion. En effet, le choix de la structure est déterminant pour la suite du projet. Le but était de pouvoir réaliser trois opérations de base sur cette file de priorité :

→ Initialisation d'une file de priorité.

→ Retirer le `teddy` de plus faible priorité de la file.

→ Placer un `teddy` dans la file en fonction de sa priorité.

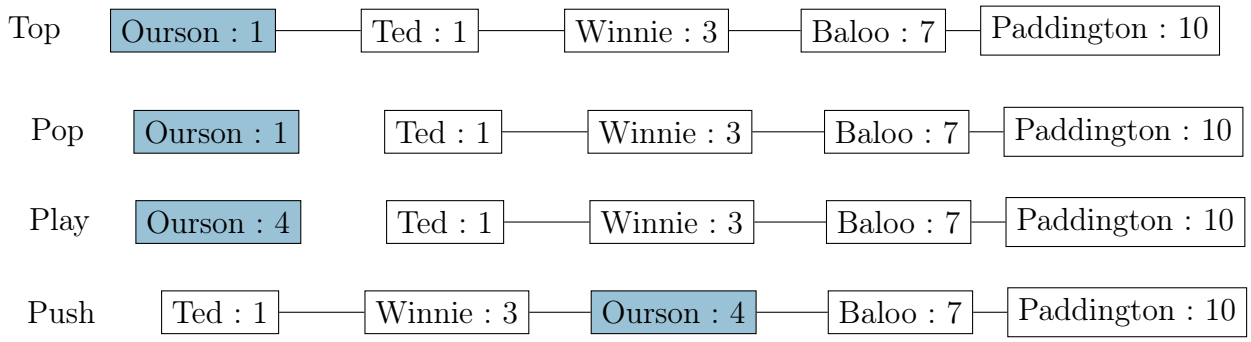


FIGURE 3 – Schématisation des opérations élémentaires sur la file

Nous nous sommes d'abord orienter vers une représentation de la file de priorité à l'aide d'un tableau de `teddys`. Néanmois, nous nous sommes rapidement rendus compte que cette implémentation était inefficace car l'insertion d'un `teddy` dans la file de priorité ne donnait pas les résultats attendus.En effet, lorsque le `teddy` est inséré dans la file, il faut également décaler tous les `teddys` qui sont dans une case d'indice supérieur ou égal à l'indice d'insertion, d'un indice vers la fin de la file. Cette opération n'était pas possible avec un tableau car la gestion de la mémoire qu'il permettait n'était pas adaptée. Nous

nous sommes alors tournés vers une représentation à l'aide d'un tableau de pointeurs, qui nous a permis de gérer ce problème. Ainsi, la structure `struct queue` (Code 4) qui représente la file de priorité contient le tableau de pointeurs vers des `struct teddy` dont la taille est égale au maximum de joueurs autorisés par le jeu (qui est défini dans le sujet par la constante MAX_PLAYERS qui vaut 20). La structure contient également un entier représentant la taille de la file.

Code 4 – Structure `queue`

```
struct queue {
    int size;
    struct teddy* tab[MAX\_PLAYERS];
};
```

La fonction `queue__create()` nous permet d'initialiser une file de priorité vide. Sa complexité en temps est constante en $O(1)$.

La fonction `queue__pop` permet de retirer le `teddy` de plus faible priorité de la file. Elle nécessite une fonction auxiliaire, nommée `queue__top` qui retourne le `teddy` placé en haut de la file (qui est celui de plus faible priorité). Nous avons choisi de maintenir le tableau de pointeurs constamment triés par ordre décroissant de priorité. Cela nous permet de réaliser la fonction `queue__top` en temps constant $O(1)$. Elle n'a donc pas d'impact sur la complexité de la fonction `queue__pop`. Cette dernière, quant à elle, est réalisé en temps linéaire. Cette complexité est due à la boucle `while` qui permet de décaler tous les `teddys` d'une case vers la tête de la file. Ce décalage est nécessaire afin de maintenir le tableau de pointeur trié dans l'ordre décrit précédemment. Cette fonction est ainsi en $O(n)$ en temps.

La fonction `queue__push` permet d'insérer un teddy dans la file de priorité en fonction de sa priorité. Ainsi, elle a recours à une fonction auxiliaire `index_of_teddy` qui retourne la valeur de l'indice de la file où insérer le `teddy`. Comme le tableau est maintenu trié, cette fonction est dans le pire des cas (celui où le teddy est de priorité plus grande que tous les autres) linéaire. La fonction `queue__push` fait donc d'abord appel à `index_of_teddy` pour stocker l'indice d'insertion du `teddy` à insérer. Ensuite, le `teddy` est inséré à l'aide d'une boucle qui est réalisée dans le pire des cas en temps linéaire. La fonction `queue__push` est donc, globalement, en $O(n)$ en temps.

3.2 Regard critique et autres choix possibles

Une alternative au tableau de pointeurs vers des `teddys` serait une liste chaînée de `teddys`. Ainsi, nous aurions utilisé une gestion dynamique de la mémoire grâce à : `malloc`, `reallocate` et `free`, qui auraient permis une gestion de la mémoire optimale. Néanmoins, encore inexpérimentés à l'utilisation des `mallocs`, nous avons préféré manipuler des pointeurs qui nous sont plus familiers.

4 Boucle de jeu

4.1 Fonctions élémentaires de jeu

Plusieurs fonctions ont été implémentées afin de faire tourner la boucle de jeu. Nous avons respecté le cadre décrit dans l'achievement 3. En effet, toute ressource différente du `HONEY` est associée à une quantité disponible maximale. De plus, tout achat alors qu'il reste moins de 10% des ressources disponibles entraîne une augmentation de la valeur de la ressource de 1 et toute action de vente alors qu'il reste plus de 90% des ressources disponibles fait baisser la valeur de la ressource de 1.

→ La fonction `parse_opts`, sert à gérer les options de jeu qui sont entrées en paramètres : le nombres de joueurs, le nombre de tours et la valeur de la graine aléatoire. Sa complexité est linéaire en la taille de `argv`. Elle est essentielle pour lancer le jeu.

→ La fonction `max_transactions_possible` prend en paramètre un pointeur vers une transaction et un pointeur vers un `teddy` et retourne le maximum de fois que ce teddy peut réaliser cette transaction. Cette fonction a une complexité en temps en $\mathrm{O}(n)$. Elle est utilisée dans la version de base du projet (achievement 0) et la première version (achievement 1). Pour la deuxième et la troisième version, nous l'avons remplacée par la fonction `transaction_possible` qui prend en paramètres un pointeur vers une transaction et un pointeur vers un `teddy`, auxquels on ajoute (pour l'achievement 3) un tableau contenant les quantités de ressources disponibles. Cette fonction retourne 0 si le `teddy` ne peut pas faire la transaction et 1 s'il le peut. Cette fonction a une complexité en temps en $\mathrm{O}(n)$, elle est linéaire en le nombre maximal de ressources.

→ La fonction `gain_transaction` retourne le gain total en `HONEY` que permet une transaction. Elle est de complexité en temps linéaire en le nombre maximal de ressources définies.

→ La fonction `best_transaction`, quant à elle, retourne la meilleure transaction possible dans une place d'échange donnée pour un teddy donné. Sa complexité en temps est linéaire en le nombre de transactions que permet la place d'échange donnée.

→ La fonction `is_visited` prend en paramètres un pointeur vers une place d'échanges et un `teddy` et retourne 1 s'il a déjà visité la place ou 0 sinon. Sa complexité temporelle est linéaire en le nombre de places visitées par le `teddy`.

→ La fonction `do_transactions` prend en paramètres un pointeur vers une transaction, un pointeur vers un `teddy` et un entier `n`, et fait faire au `teddy` `n` fois la transaction donnée.Sa complexité est linéaire en le nombre maximal de ressources. On l'utilise pour l'achievement 0 et l'achievement 1. Pour l'achievement 2, on lui apporte des modifications. En effet, le paramètre `n` disparaît et le `teddy` ne fait qu'une seule fois la transaction passée en paramètre. Enfin, pour l'achievement 3, on lui passe en paramètre supplémentaires un tableau qui contient les quantités disponibles de ressources. Elle permet alors

au `teddy` de réaliser une fois la transaction passée en paramètres et modifie les quantités de ressources disponibles.

Ces fonctions permettent de réaliser les actions nécessaires au déroulement du jeu et servent de base à l'implémentation des stratégies.

4.2 Stratégie de base : aléatoire

4.2.1 Implémentation

Dans un premier temps, nous allons implémenter une stratégie totalement aléatoire. Dans cette version, qui correspond à l'achievement 0 et à l'achievement 1, les `teddys` jouent de manière aléatoire en sélectionnant une place d'échange et en échangeant des ressources sur cette place un nombre aléatoire de fois, en fonction de leur moyens. Ensuite, losqu'il a réalisé sa transaction il est redirigé vers la place d'échange suivante. La fonction `active_teddy_play1` implémente cette stratégie. Elle fait appel à des fonctions auxiliaires définies précédemment : `is_visited`, `max_transactions_possible` et `do_transactions`. Dans le pire des cas, sa complexité en temps est en $\mathrm{O}(n)$.

4.3 Stratégie 2 : Choix de la transaction la plus rentable

4.3.1 Principe

Dans cette amélioration, nous implémenterons une nouvelle stratégie qui consiste à choisir la transaction qui permet le plus gros gain positif et qui donne la possibilité au `teddy` de passer son tour et de changer de place d'échanges si aucune transaction ne lui permet un gain positif.

4.3.2 Fonction de jeu adaptée

La fonction `active_teddy_play2` permet d'implémenter cette stratégie. En effet, elle fait appel aux fonctions `do_transactions` et `best_transaction`. Cette fonction a une complexité en temps en $\mathrm{O}(n)$.

4.4 Stratégie 3 : Aléatoire intelligent

4.4.1 Principe

Nous avons choisi d'implémenter une troisième stratégie qui consiste à jouer de manière aléatoire mais en ne retenant que les transactions de gain positif.

4.4.2 Fonction de jeu adaptée

La fonction `active_teddy_play3` permet de réaliser cette stratégie. Elle fait appel aux fonctions : `do_transactions` et `best_transaction`. Cette fonction a une complexité en temps en $\mathrm{O}(n)$ où n est le nombre maximal de ressources.

4.5 Comparaison des différentes stratégies

Nous avons collecté des résultats de tests, grâce à un script Shell, afin de comparer les différentes stratégies entre elles et de trouver la plus efficace. La stratégie 2 présente 48% de réussites tandis que la stratégie 3 présente 52% de réussites. La stratégie 1 ne permet pas le gain, ce qui est cohérent.

4.6 Autres stratégies envisageables

Nous n'avons pensé, pour l'instant, qu'à des stratégies où le `teddy` ne considère que les transactions de la place dans laquelle il se trouve pour établir sa stratégie. Il serait judicieux d'implémenter une stratégie algorithmique qui calculerait le gain permis par les différents chemins du graphe des places déchanges et de choisir le plus avantageux. Cette stratégie serait une stratégie gloutonne, assez coûteuse en temps.

Il serait intéressant de considérer une solution qui se baserait sur du Machine Learning. En effet, cela commencerait par l'implémentation d'un réseau de neurones adapté à la situation. Ensuite, il faudrait procéder à une étape d'apprentissage, en laissant le `teddy` faire de nombreuses parties en gardant pour objectif d'avoir le plus gros gain. Mais cette stratégie est difficile à implémenter et nécessite une grande dimension théorique.

5 Conclusion et Discussions

Pour conclure, ce projet nous a permis de découvrir la manière de travailler de manière collaborative (utilisation de git) sur un même code. De plus, nous avons pu mettre en pratique et approfondir nos connaissances en matière de programmation `C`, en environnement de travail, gâce à l'utilisation des commandes bash et également en algorithmique théorique. En effet, nous avons sans cesse questionné la complexité en temps de nos algorithmes.

Il aurait été intéressant de développer une interface graphique pour le jeu qui a un fort aspect ludique. En effet, on peut l'assimiler à un jeu de plateau où le plateau est composé de plusieurs places d'échanges et les `teddys` seraient représentés par des pions.