BORDEAUX INP Enseirb-Matmeca

Rapport Projet 1 – Munificence

Nemo D'ACREMONT, Martin EYBEN

Sujet : Munificence

S5 - Année universitaire 2023-2024

Contents

Préambule

Ce projet avait pour objectif la mise en oeuvre de notions étudiées tout au long du premier semestre, à la création d'un jeu plus ou moins inspiré du jeu de société "Splendor".

Il s'agissait dans un premier temps de mettre en place les fondamentaux du jeu, puis de les étendre plus ou moins artificiellement, nous forçant à faire preuve de rigueur dans nos méthodes et à mobiliser nos connaissances algorithmiques de façon à aborder sereinement les problèmes que nous rencontrions.

I Organisation du projet

1 Organisation du travail en équipe

Problématique

Le travail en équipe n'est pas une chose évidente, et il est nécessaire de mettre en place une méthode afin d'optimiser la productivité, sinon le risque qu'un même sujet soit traité par deux personnes existe, ce qui entrave l'avancement.

Solutions mises en place

Méthode Kanban

Nous avons utilisé l'application web kanboard, installée sur un serveur personnel, afin de distribuer le travail à faire, ainsi nous savions à tout moment, ce qu'il y avait à faire, ce qui était en train d'être fait et ce qui avait été fait.

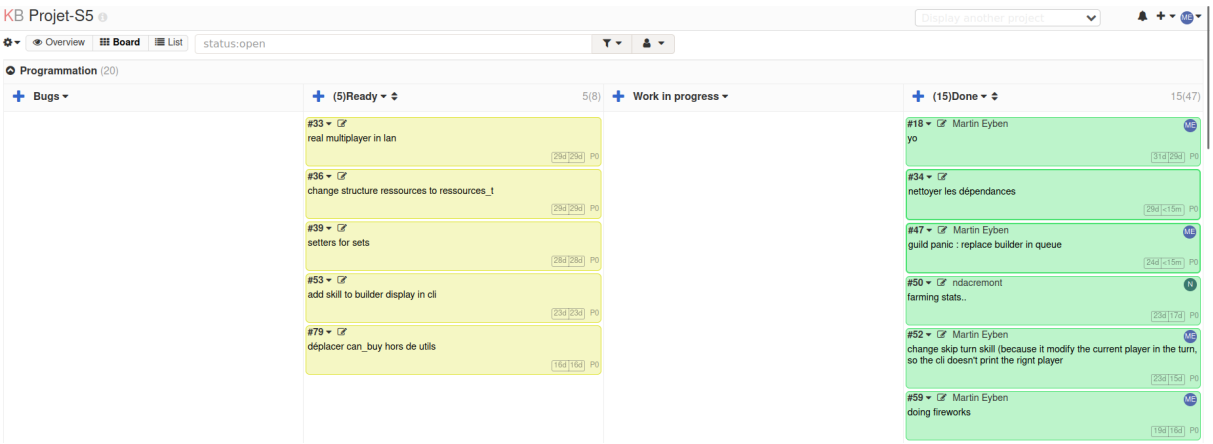


Figure 2: Utilisation de Kanboard

Utilisation de git

L'utilisation d'un gestionnaire de version est essentiel pour la réalisation de ce projet. Cependant, se limiter à une utilisation élémentaire de ce logiciel pour le travail en équipe peut mener à une multiplication de problèmes de conflits, pouvant entraver l'avancée du projet.

Nous avons opté l'utilisation de 3 branches : la branche master, qui se devait d'être propre, le code qui s'y trouvait devait toujours être compilable, et devait contenir le moins de bugs possible. Les deux autre branches que nous utilisions nous étaient attitrées, à chaque fois que nous nous attribuions une tache sur le kanboard, nous développions une solution sur notre branche puis nous fusionnions sur la branche master.

Schématiquement, notre utilisation de git se résume au schéma suivant :

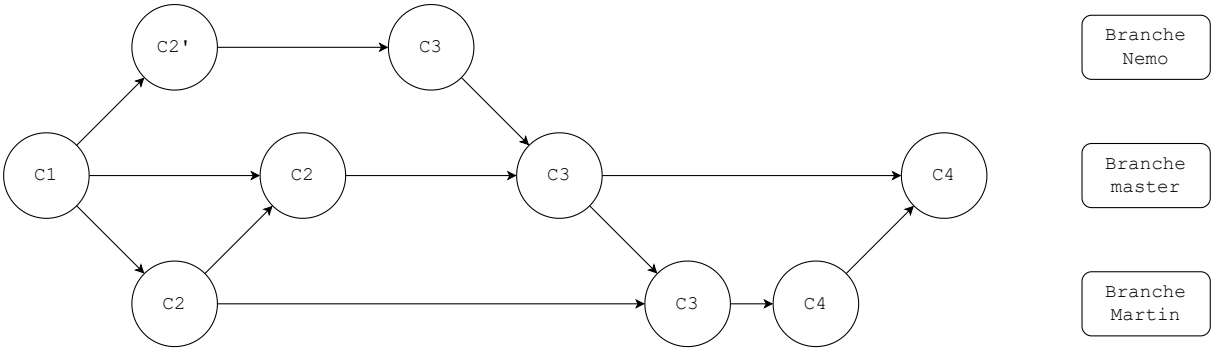


Figure 3: Utilisation de nos 3 branches de git

2 Organisation du code

Problématique

Un projet d'une telle envergure nécessite une organisation spéciale afin d'être mené à bien. Il s'agit d'une organisation qui doit faciliter l'ajout de nouvelles fonctionnalités, faciliter la modification de fonctionnalités déjà présentes et faciliter la lecture et la compréhension de ce qui a déjà été fait.

Solutions mises en place

Division en dossiers principaux

Sachant que nous nous apprêtions à créer un exécutable pour le projet en lui-même, pour les tests, l'évaluation de partie et l'interface graphique, nous avons décidé de séparer les sources pour chaque exécutable dans un dossier séparé.

Ainsi, notre projet est constitué de 4 dossiers principaux: `/src`, `/tst`, `/cli_src`, `/evaluator_src`, comme le schématise le schéma ci-dessous :

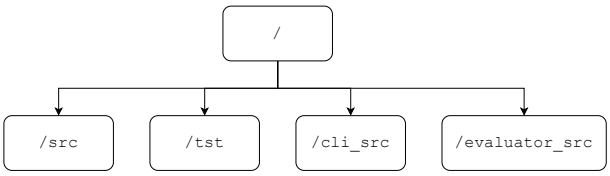


Figure 4: Schéma des dossiers principaux du projet

Division du code par thème

Une fois divisé en dossiers principaux, le code est re-divisé en sous-dossiers au sein de ces dossiers. Ainsi, on retrouve un sous-dossier pour les architectes et la structure de guilde, un pour les jetons et la structure de marché etc... Ainsi qu'un sous-dossier `/src/utils`, qui va contenir toutes les structures et fonctions génériques, comme une structure de file ou une *macro* `MIN` qui retourne le minimum entre 2 entiers.

L'architecture des sous-dossiers de `/src` est schématisé ci-dessous :

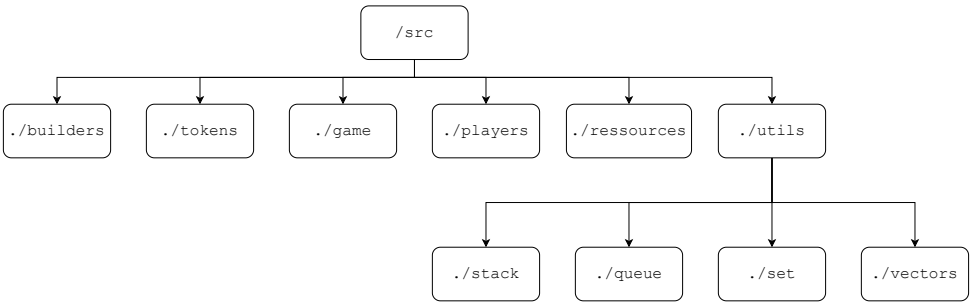
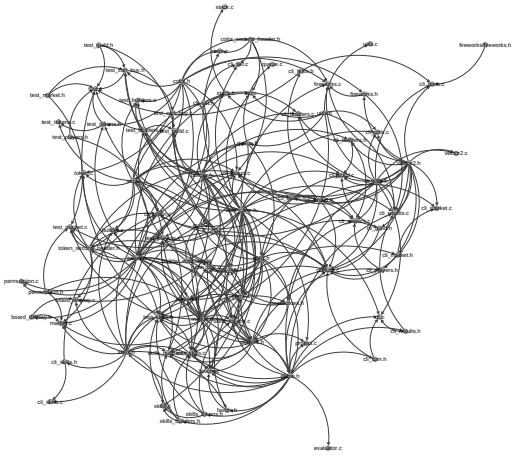


Figure 5: Schéma sous-dossiers de */src*

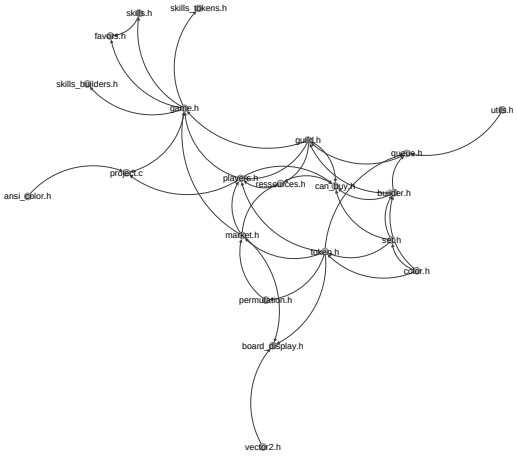
Les dépendances

Étant donné le nombre de fichiers présent dans le projet, nous avons essayé de générer automatiquement un graphe des dépendances. L'utilisation l'option `-MM` de `gcc` créé des connexions entre les fichiers malgré le fait qu'il n'y ait pas de dépendances directes.

Nous avons donc opté pour l'écriture d'un script python qui lit tous les fichiers finissant par `.[ch]` du projet, et lisant chacune des inclusions pour déterminer les dépendances directes de chaque fichier.



(a) Graphe des dépendances incluant tous les fichiers



(b) Graphe des dépendances simplifié

Mise en place d'une convention

Problématique

Le travail en groupe implique de devoir relire le code. Ainsi, la forme du code requiert une certaine attention afin de rendre l'utilisation de ces outils et la lecture du code naturelle.

Solution mise en place

Pour résoudre ce problème de forme, une convention de nommage a été mise en place pour les variables et les fonctions, ainsi que des règles d'écriture.

Nous avons décidé d'utiliser la convention `snake_case` pour ce qui est de la forme, nous nommons les types `struct type_t` afin de ne pas les confondre avec d'éventuelles variables. Une fonction qui devait s'appliquer à un type `struct type_t` en particulier est notée `type_function(...)`.

Ci-dessous un exemple non-exhaustif de la mise en pratique de ces conventions

Listing 1: Exemples de mise en pratique de ces conventions

```
struct queue_t ;


unsigned int queue_append(struct queue_t* queue, void* value);
```

Compilation

Problématique

Travailler avec telle structure de fichier rend la compilation moins évidente, et fastidieuse si on voulait la faire manuellement avec `gcc`.

Solution mis en place

Nous avons utilisé `make`, et nous nous sommes appliqué à l'écriture d'un `makefile` qui permettrait d'avancer sereinement dans le projet.

Notons avant tout que notre `makefile` est largement inspiré des ébauches proposées du *blog* de Job Vranish. Ces ébauches formaient une fondation solide sur laquelle travailler. Cependant, nous nous le sommes approprié afin de l'adapter à notre structure de développement.

Comme la plupart des sources du dossier `/src` sont partagées entre chaque exécutables, nous avons fait le choix de d'abord chercher tous les fichiers sources de `/src`, puis de retirer le fichier `/src/projet.c` contenant la fonction main.

Listing 2: Filtrage des fichiers sources du jeu

```
SRC_DIRS := ./src
PROJECT_MAIN_FILE_NAME := ./src/project.c

SRCS := $(shell find $(SRC_DIRS) -name '*.c')
SRCS := $(filter-out $(PROJECT_MAIN_FILE_NAME), $(SRCS))
```

Nous avons opté pour l'utilisation de l'option `-Isous_dossier` de `gcc`, permettant de déclarer des librairies systèmes. Ainsi, en l'utilisant avec tous les sous-dossiers de `/src` et des autres dossiers principaux, cela nous permet de nous limiter à l'écriture de '`#include "fichier.h"`' dans nos fichiers sources plutôt que le chemin relatif vers le fichier `fichier.h`. Étant donné l'arborescence complexe, cela simplifie largement la lecture et l'écriture des dépendances.

Cependant, il fallait aussi que nos éditeurs, par le biais de clang, puissent être utilisable. D'où l'ajout de la cible `clang_custom_lib_support` dans le makefile, permettant de créer le fichier de configuration `compile_flags.txt` indiquant nos librairies systèmes personnalisées à `clang`.

Finalement l'utilisation d'un nouveau dossier, le dossier `/build`, sert à stocker tous les fichiers de compilation. Pour cela, nous recréons la structure précédente du projet, comme le schématise la figure 7.

L'utilisation de ce dossier permet d'avoir une séparation claire entre le reste du projet et la partie compilation.

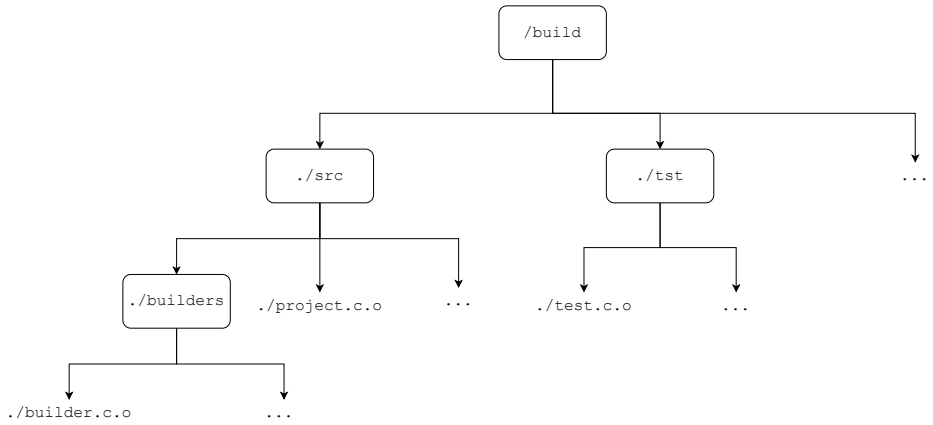


Figure 7: Schéma arboressence du dossier /build

II Présentation du projet

1 Le splendor

Le splendor est un jeu de société se jouant en tour par tour, mettant en jeu des jetons de couleurs, des architectes, et dont le but est d'obtenir le plus de points possibles. Les jetons apportent des ressources permettant ensuite de recruter des architectes. Les architectes, quant à eux, permettent de générer des ressources à chaque tours et rapportent des points.

Les architectes sont stockés dans une guilde, et sont disponibles par niveaux. Pour chacun des niveaux, seuls un nombre prédéterminé sont disponible. Lors de l'achat d'un architecte, on le remplace par le prochain disponible de la pile du niveau associé, comme le schématise la figure 8 suivante :

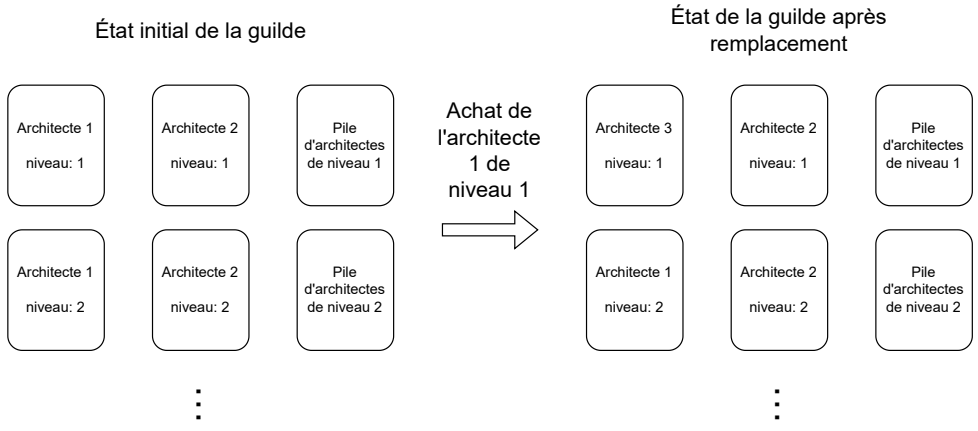


Figure 8: Schéma de l'achat et remplacement d'un architecte de niveau 1

Les jetons sont eux stockés dans un marché et sont disposés sur un plateau carré. Pour être récupérés, les jetons doivent être pris connexes par rapport au chemin.

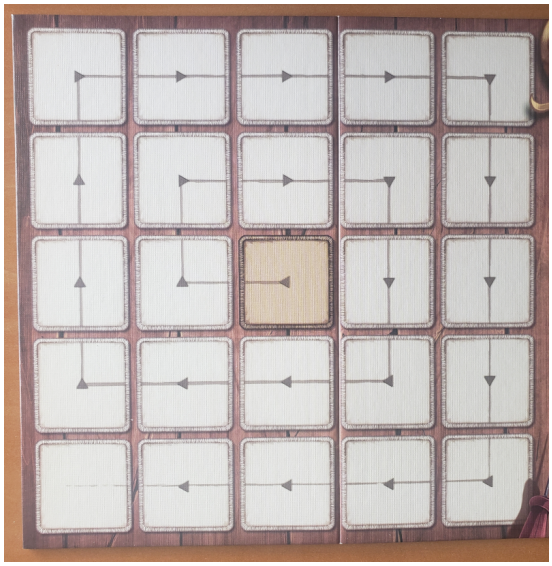


Figure 9: Images schématisant le plateau et le chemin reliant les jetons

À chaque tours, il n'est possible de faire qu'une action, on peut choisir de soit prendre entre 1 et 3 jetons dans le marché, soit d'embaucher un architecte auprès de la guilde. Lors de la prise de jetons ou de l'embauche d'un architecte, il est possible qu'un pouvoir soit attaché à ce dernier, et alors le pouvoir s'exécute à la fin de l'action.

Remarque 1

Au début d'un tour, il est aussi possible, si on a accumulé une faveur, de l'utiliser pour prendre un jeton dans le marché ou pour renouveler les architectes disponibles d'un niveau donné de la guilde.

2 Contraintes du projet

En plus de devoir implémenter, nous devions faire attention aux contraintes suivantes du sujet :

- Ne pas modifier les fichiers `color.h`, `builder.h` et `token.h` qui étaient fourni.
- Utiliser exclusivement le langage C.
- Le fichier `makefile` devait définir une règle `project`, qui devait créer l'exécutable nommé `project` à la racine du projet, et la règle `all` devait faire appel à la règle `project`.
- Le fichier `makefile` devait aussi définir une règle `test`, qui devait exécuter les différents tests du projet.
- La définition de `NUM_LEVELS` et `NUM_TOKENS` doit être possible lors de l'appel de la règle `project`

3 Nos contraintes

A titre d'expérimentation, nous nous sommes aussi contraint à ne pas utiliser de `malloc`.

Le code sera entièrement écrit en anglais et on privilégiera l'utilisation de pointeur dans les paramètres de nos fonctions.

III Implémentation du jeu

1 Les architectes, jetons et couleurs

1.1 Les architectes

La contrainte de non-modification du fichier `builder.h` nous force à implémenter les architectes comme un type abstrait.

Ainsi, un architecte ne peut être déclarer dans un autre fichier qu'en proposant une nouvelle définition du type `builder_t`, nous avons plutôt décidé de déclarer au début de la partie `num_builders` architectes, et de ne travailler par la suite que sur des pointeurs de ces-dits architectes et les primitives définies dans `builder.h`.

1.2 Les jetons

De même que pour les architectes, nous ne pouvions pas modifier le fichier `token.h`, cependant, l'implémentation du type jeton n'est pas abstrait, il est donc possible de créer des jetons n'importe où dans le projet

Nous avons cependant décidé, comme pour les architectes, de créer au début de la partie `NUM_TOKENS` jetons, puis de les manipuler par l'intermédiaire de pointeurs. Cela a l'avantage d'avoir un nombre maîtrisé de jetons tout au long de la partie.

1.3 Les couleurs

Dans le jeu, `NUM_COLORS` couleurs sont en jeux, avec un maximum de 10 couleurs. Celles-ci sont codées comme des entiers à l'aide du type `enum color_t`.

2 Les guildes et les marchés

2.1 Les guildes

Nous avons implémenté les guildes dans un premier temps pour modéliser les achats possibles du jeu, mais nous avons étendu son utilisation au stockage d'architectes en général. Ainsi le type de guilde est utilisé pour que les joueurs puissent acheter des architectes, mais aussi utilisé par les joueurs pour stocker les architectes qu'ils ont pu acheter.

Notre structure de guilde se présente de la manière suivante :

Listing 3: Implémentation du type struct guild_t

```
struct guild_t
{
        struct builder_t* builders[MAX_BUILDERS];
        int n_builders;
        struct queue_t available_queue[NUM_LEVELS];
        struct available_builders available_builders;
```

```
};
```

Le tableau `builders` permet de stocker les pointeurs des architectes présent dans la guilde, `n_builders` le nombre d'architectes présents dans la guilde, `available_queue`, bien qu'étant une file, permet de modéliser la pile des architectes qui ne sont pas encore achetable, et `available_builders` stocke les architectes disponibles à l'achat.

Remarque 2

Nous avons décidé d'utiliser une file pour stocker les prochains architectes plutôt qu'une pile car cela permet de faire cycler plus facilement les architectes, et la caractéristique d'une pile, de remettre un architecte en haut de cette dernière, n'est jamais utilisée dans le projet.

2.2 Les marchés

Les marchés servent avant tout à stocker les jetons. Un marché global permettant aux joueurs de piocher des jetons est créé, puis chaque joueur peut utiliser son propre marché pour ensuite stocker ses jetons.

Nous définissons le type `struct market_t` de la manière suivante :

Listing 4: implémentation du type struct market_t

```
struct market_t {
        struct token_t* tokens[NUM_TOKENS];
        struct permutation_t permutation;
};
```

Nous utilisons le tableau `tokens` pour stocker les pointeurs de jetons présent initialement dans le marché global. `permutation` permet d'appliquer une permutation sur le replacement des jetons dans le marché (utilisé pour le marché global).

Particularité du marché global

Les joueurs doivent pouvoir piocher uniquement des jetons qui sont connexes. Pour cela la fonction `market_get_linked_tokens` permet de renvoyer l'indice du premier jeton d'un groupe de nb-jetons connexes du marché global (cf 1).

3 Les joueurs

Les joueurs sont essentiellement une structure capable de stocker des architectes et des jetons. Pour cela on décide de les implémenter de la sorte :

Listing 5: Implémentation des joueurs

```
struct ressources_t
{
        struct market_t market;
        struct guild_t guild;
};

struct player_t
```

```
{
        struct ressources_t ressources;
        int current_point;
        unsigned int favor;
        unsigned int id;
};
```

Chaque joueur possède sa guilde et son marché ainsi que son nombre de point et le nombre de faveur qu'il possède. De cette manière on peut implémenter les méthodes liées aux joueurs à l'aide des fonctions liées aux marchés et aux guildes.

4 Les pouvoirs et faveurs

4.1 Pouvoirs

Problématique

Pour implémenter les pouvoirs, il est nécessaire qu'ils partagent tous la même signature pour que l'on puisse stocker les adresses des fonctions avec les jetons / architectes. Par ailleurs il n'est pas possible de modifier la structure des architectes et des jetons, il faut donc réfléchir à un autre moyen de les relier.

Implémentation des pouvoirs

Les pouvoirs partagent donc la même signature qui contient le tour actuel (cf 5) et ce qui à provoquer l'exécution du pouvoir. On définit un nouveau type `skill_f`:

Listing 6: Signature des pouvoirs

```
typedef int (*skill_f)(struct turn_t* turn, const void* trigger);
```

Le développement des pouvoirs se faire alors aisément à l'aide des nombreuses sous fonctions, un pouvoir étant une suite d'actions qui auraient pu être exécuté lors d'un tour (un joueur étant composé d'un marché et d'une guilde (cf 3), les interactions entre joueurs deviennent des interactions avec un marché ou une guilde).

Le pouvoir `Main de maître` a demandé plus d'attention. En effet, il est nécessaire de filtrer les jetons du marché pour ne récupérer que les jetons qui ont une intersection avec ce que procure l'architecte. Pour cela on a créé une fonction capable de retourner l'intersection de deux `set_t`.

Comment lier les pouvoirs aux jetons / architectes

Stockage des pouvoirs

Pour ne pas avoir à modifier la structure des jetons et des architectes, nous avons décidé de recréer une sorte de dictionnaire. On associe une adresse (de jeton ou architecte), à un tableau d'identifiants de pouvoirs contenant au plus `MAX_SKILLS_PER_TRIGGER`.

Pour cela on initialise un tableau en statique avec une structure contenant le couple (`void*` , `enum skills_id skills[MAX_SKILLS_PER_TRIGGER]`).

De cette manière on peut récupérer les pouvoirs à l'aide de la fonction suivante qui parcourt le tableau à la recherche du pointeur.

Listing 7: Récupération des pouvoirs

```
enum skills_id* skills_get_by_trigger(const void* trigger);
```

Exécution des pouvoirs

On peut ensuite exécuter les pouvoirs associés en récupérant les pointeurs de fonction associé à l'id du pouvoir à l'aide de la fonction `skill_exec`.

4.2 Les faveurs

Les faveurs jouissent de la même implantation des pouvoirs, mais sont stockés dans une `enum` différente pour permettre leur exécution en début de partie.

Listing 8: Enumération des faveurs

```
enum favor_id {
        NO_FAVOR,
        FAVOR_RETURN,
        FAVOR_RENEWAL,
        NUM_FAVOR
};
```

5 Le jeu

Structure d'une partie

Problématique

Le nerf du projet se trouve dans l'implémentation d'une partie. Elle doit permettre de jouer chaque tour séparément, connaître l'état du tour précédent et ainsi pouvoir avoir un historique de la partie. Ce qui sera notamment très utile lors de la création de l'interface mais aussi pour pouvoir jouer de manière indépendante plusieurs parties. (cf VII et VIII)

Architecture

Structure de tour

Listing 9: Implémentation de la structure turn_t

```
struct turn_t
{
        struct market_t market;
        struct guild_t guild;
        struct player_t players[MAX_PLAYERS];
        unsigned int current_player;
        unsigned int points_to_win;
```

```
        unsigned int display; /* Used to display in other functions*/
        unsigned int num_player;
        unsigned int id;
        struct game_parameters params;
        struct context context;
};
```

Chaque tour possède la copie de la partie à un instant t.

On y retrouve l'état du marché, de la guilde et les inventaires des joueurs à l'issue du tour. Mais également le contexte de ce qu'il s'est passé dans le tour (cf VIII).

Chaque tour possède volontairement beaucoup d'informations sur la partie car cela va permettre de jouer énormément avec ces paramètres lorsque l'utilise en paramètre de fonction (pour les faveurs et les pouvoirs notamment, cf 4.1).

Structure de partie

Listing 10: Implémentation de la structure game_t

```
struct game_t
{
    // +1 for the init state, +1 for the final state
    struct turn_t turns[MAX_MAX_TURNS + 1 + 1];
    unsigned int num_turns;
    unsigned int current_turn_index;
};
```

Chaque partie stocke l'ensemble des tours qui ont été joués, et contient au plus `MAX_MAX_TURNS` tours (toujours pour éviter l'utilisation de `malloc` cf 3).
num_turns indique le nombre de tours maximum de la partie (par défaut 10 mais peut être spécifié avec le paramètre -m) et `current_turn_index` l'indice du tour actuellement joué.

Fonctionnement d'une partie

Initialisation de la partie

Pour initialiser une nouvelle partie, on utilise la fonction `init_game` avec les paramètres souhaités.

```
/*
        Init game with params
*/
void init_game(struct game_t*, struct game_parameters);
```

Listing 11: Structure des paramètres de la partie

```
struct game_parameters
{
        unsigned int max_turns;
        unsigned int points_to_win;
        unsigned int builder_seed;
        unsigned int market_seed;
        unsigned int random_seed;
```

```
        unsigned int display;
        unsigned int num_player;
};
```

La fonction `init_game` modifie le premier tour de la partie en initialisant toutes les instances et en imposant les paramètres de la partie.
Le tour est alors sauvegardé (cf 13) et on peut lancer la partie avec `play_game`.

Particularité de `rand`

La fonction `rand` étant non ré-entrante, il s'agit de l'unique endroit où `srand` est appelée. `srand` est appelée une fois lors de l'initialisation des architectes (cf 1.1) et une fois lors de l'initialisation des jetons (cf 1.2) afin de permettre la création de decks aléatoires indépendants.
`srand` est alors appelée une dernière fois avec `random_seed` pour initialiser le hasard du reste des actions prises dans la partie.

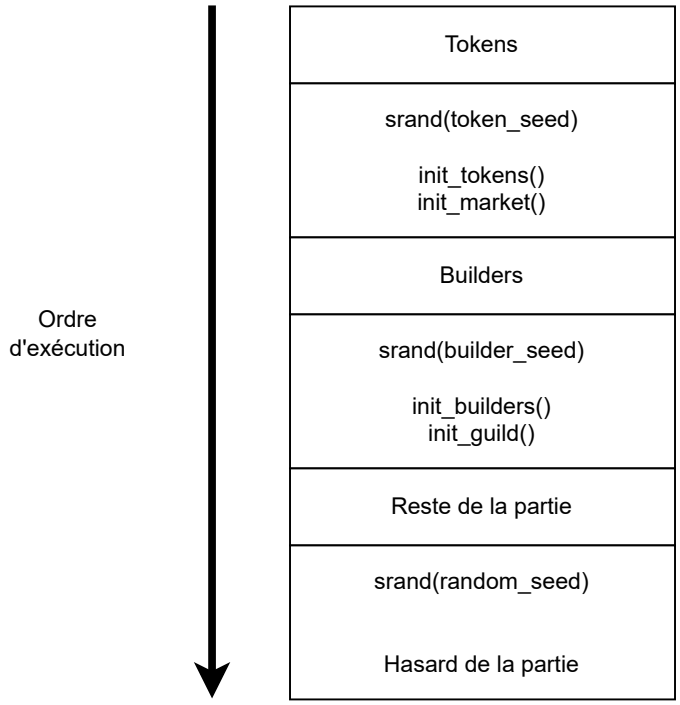


Figure 10: Utilisation de `srand`

Exécution d'un tour

Listing 12: Implémentation de l'exécution d'un tour

```
struct turn_statistics turn_play(struct turn_t* current_turn)
{
    /* Favors execution */

        /*
        Take a random decision and check if it's possible to hire a
            builder
        */
        unsigned int random_choice = rand() % 100;
```

```
        struct builder_t* builder_to_buy = select_affordable_builder(
            guild, current_player);

        if ((random_choice < 50) && (builder_to_buy != NULL))
        {
                /* Hire builder and execute associated skills */
        }
        else if (random_choice < 90)
        {
                /* Pick tokens and execute associated skills  */
        }
        else
        {
                /* Skip turn */
        }

}
```

A l'aide de la fonction `turn_play`, le tour qui est passé en paramètre est joué.

On s'occupe dans un premier temps des faveurs (cf 4.2) puis on joue le reste du tour. Pour cela on récupère le premier architecte achetable (cf 2) et on décide d'une action.
Le joueur a :

- 50% de chance de recruter un architecte (s'il n'en a pas la possibilité, il pioche des jetons)
- 40% de chance de piocher des jetons
- 10% de chance de passer son tour

Lorsque le joueur pioche des jetons ou recrute un architecte, on exécute ensuite les pouvoirs éventuellement associés à ce ou ces derniers à l'aide de `skill_exec` (cf 4.1).

On finit par ajouter les actions aux statistiques et au contexte du tour (cf VII et VIII).

Sauvegarde d'un tour

Listing 13: Sauvegarde d'un tour

```
void game_save_turn(struct game_t* game)
{
    /*  things before */
        memcpy(game_get_turn(game, current_turn_index + 1),
            game_get_current_turn(game), sizeof(struct turn_t));

    /* change other params */
}
```

Lorsque qu'un tour est joué (à l'aide de `play_turn`) on copie l'état actuel de la partie dans la prochaine case du tableau `turns` de la structure `game`. Ainsi le tour à la case i du tableau correspond à l'état de la partie à l'issue du i-eme tour.
On en profite pour modifier les paramètres du prochain tour qui doivent l'être.

Boucle de jeu

Listing 14: Implémentation de la boucle de jeu

```c
struct game_statistics game_play(struct game_t* game)
{
    /*
        Game loop
    */
    while (!has_won(current_turn) && current_turn_index <= num_turns)
    {
        struct turn_statistics turn_stats = turn_play(current_turn);

        /* Switch to the next turn */
        game_save_turn(game);
        next_player(game_get_current_turn(game));
    }
}
```

La boucle de jeu consiste à jouer des tours tant que la partie n'a pas été gagné et que l'on n'ai pas atteint le nombre maximum de tours. On passe au tour suivant en sauvegardant l'état actuel et en changeant de joueur.

IV Tests

Problématique

Lorsque que le code est important et est amené a évoluer avec de nouvelles fonctionnalités, il peut devenir intéressant de tester le comportement des fonctions pour vérifier qu'elles respectent des règles de base. Dans notre cas de figure nous testons une grande majorité des fonctions de base, les autres fonctions découlant de ces dernières.

1 Mise en place des tests

Tout comme le reste du code, les tests sont séparés en sous dossiers par thème. Chaque thème possède une fonction qui regroupe les tests des fonctions associées à ce thème, qui sont alors exécutés par l'exécutable de test.

Listing 15: Exécution des tests

```c
int main(int argc, char *argv[])
{
        test_token();
        test_builders();
        test_market();
        test_guild();
        test_players();
        test_utils();
        test_skills();

        return EXIT_SUCCESS;
}
```

2 Vérifications effectuées

Chaque tests vérifie le comportement attendu de la fonction, et si elle respecte les règles définies en amont.

Prenons l'exemple du test de rendu de monnaie, on se place dans le cas le plus défavorable pour notre algorithme : un prix avec plusieurs couleurs, avec de nombreuses ressources pour payer. On teste la fonction sur ce cas particulier.

On effectue les étapes suivantes :

- On essaye de payer le prix avec des jetons simples
- On retire un jeton au hasard et on vérifie qu'on ne peut plus payer
- On réessaye de payer mais cette fois ci avec des jetons complexes
- On vérifie que le rendu de monnaie est le meilleur
    * Pour cela on ajoute au marché de test un jeton complexe avec exactement le prix et on vérifie qu'il s'agit bien du seul jeton retourné par l'algorithme

V Solutions algorithmiques

1 Récupérer des jetons connexes d'un marché

Pour récupérer nb-jetons connexes d'un marché, on parcourt ce dernier du début à la fin avec un compteur de jetons connexes.
Différents cas de figure :

- Si le pointeur n'est pas `NULL` (la case du marché contient un jeton) :
    - Si le compteur est égal à nb alors on ajoute l'indice à une liste sans incrémenter le compteur.
    - Sinon on incrémente le compteur de 1
- Si le pointeur est `NULL` (la case du marché est vide) :
    - On remet le compteur à 0.

Ainsi à l'issue de la boucle on peut retourner un indice présent dans la liste pris de manière aléatoire. Si la liste est vide, on retourne -1.

La complexité de cet algorithme est $\theta(n)$, avec n le nombre de jetons dans la partie.

Remarque 3

Ici on considère des jetons connexes lorsqu'ils sont reliés par le chemin continu dessiné par le plateau (cf 9) et non lorsqu'ils sont voisins sur le plateau. C'est un choix assumé.

2 L'achat d'un architecte

2.1 Algorithme glouton

L'algorithme glouton va chercher si on est capable de payer le prix et renvoyer la première façon de le faire.

Pour cela on retire au prix à payer la production de tous les architectes, puis va parcourir l'ensemble des jetons du joueur. Dès qu'un jeton peut être utilisé pour payer le prix, on retire au prix ce que procure le jeton et on l'ajoute au marché utilisé pour payer (qui est retourné à l'issu de l'algorithme).

Listing 16: Algorithme glouton de rendu de monnaie

```
procedure rendu_glouton(var guilde : Guilde; var ressources : Marche;
    prix : Set);
var
    i: integer;
    jeton : Jeton;
    out : Marche;
begin
    for i := 0 to MAX_BUILDER do
    begin
        prix := retirer_au_prix(prix, builder[i].provide)
    end;

    for i := 0 to NUM_TOKENS do
    begin
        jeton := ressources[i]
        if utilisable(prix, jeton) then
        begin
            ajouterJeton(out, jeton);
        end;
        if prix == set_nul then
        begin
            retourner out;
        end;
    end;

    return set_nul;
end;
```

La complexité de cet algorithme est linéaire par rapport au nombre maximum de jetons.

2.2 Algorithme récursif

Contrairement à un algorithme glouton, l'algorithme récursif va permettre de récupérer la meilleur façon de payer un architecte.

Tout comme l'algorithme glouton, on commence à retirer au prix ce qui est produit par les architectes. On teste ensuite l'ensemble des combinaisons de jetons (stocké dans un marché) pour payer le prix.

A chaque fois que l'on peut utiliser une combinaison pour payer le prix, on compare cette combinaison avec la précédente pour sélectionner la meilleure (cf 3). On teste l'ensemble des des combinaisons de nb-jetons avec nb variant entre 1 et le nombre de ressrouces dans le prix à payer.

Avec les k-uplets suivants représentant l'ensemble des jetons présents dans le marché testé, l'arbre des appels du rendu de monnaie est construit comme suivant :

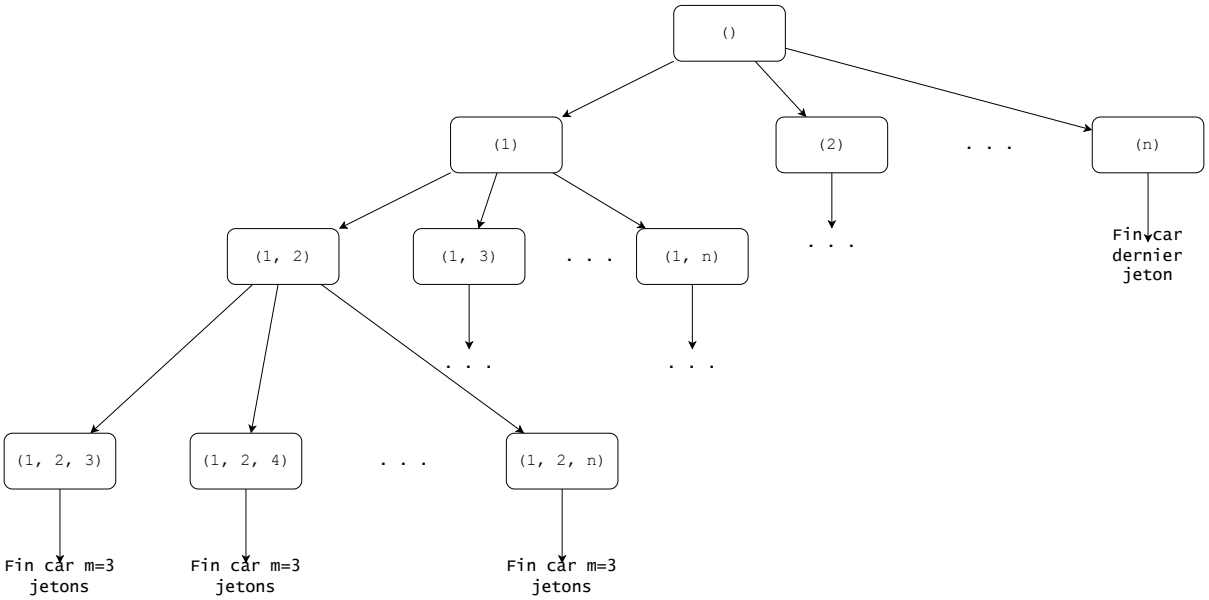


Figure 11: Arbre des appels du rendu de monnaie

L'algorithme pour tester les combinaisons de nb-jetons ressemble à cela :

Listing 17: Algorithme de test des combinaison de nb-jetons

```
void find_max_eff_sub_market_rec(args)
{
    if (est_utilisable(marche_teste))
        if (non est_utilisable(meilleur))
            meilleur = marche_teste;

        else
            *meilleur_marche = max_marche(meilleur_marche,
               marche_teste);

    else if (k < m)
    {
        for (chaque appel du dernier indice a n)
            /* Appel recursif */;
    }
}
```

Complexité, terminaison et correction

Complexité

L'algorithme de rendu de monnaie teste l'ensemble des combinaisons de jetons.

Or :

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n \tag{1}$$

avec n le nombre de jetons maximum pour un joueur. Tester si un marché est utilisable se fait en complexité linéaire.

On a donc une complexité exponentielle en $\Theta(n2^n)$.

Remarque 4

Cependant le prix maximum ne possède jamais autant de couleur que le nombre de jeton que possède un joueur. Dans notre cas de figure, le prix se limite à $m = 3$ couleurs différentes.
Ainsi on ne va tester seulement $\sum_{k=1}^{m} \binom{n}{k}$ combinaisons de jetons. La complexité de l'algorithme est donc grandement diminué et on a plutôt un algorithme en $O(n^m)$ lorsque m ne dépend pas de n, ce qui est le cas dans notre version actuelle du jeu.

Terminaison

À chaque appel récursif, k augmente de 1, donc en considérant $m = resources\_requises$, on peut poser un variant $V = m - k$, qui un entier naturel strictement décroissant.

On a donc la terminaison du rendu de monnaie.

Correction

On note :

- a_acheter est l'ensemble de ressource qu'on essaie d'acheter
- eff(marché): float une fonction qui associe à un marché une efficacité, le but est de la maximiser
- est_utilisable(marché): booléen retourne vrai si marché permet de payer a_acheter
- "meilleur" le meilleur marché jusqu'à présent au sens de eff
- max_marche(marché1, marché2): marché retourne le meilleur marché entre marché1 et marché2 au sens de eff

On pose :

- PRECOND = "non est_utilisable(`meilleur`) $\Rightarrow$ `meilleur` $= \emptyset$"
- POSTCOND = "si est_utilisable(`meilleur`) alors eff(`meilleur`) = max(efficacité des sous-marchés utilisables, sinon meilleur $= \emptyset$"

Démonstration par induction

Si PRECOND,

Cas de base :

Si est_utilisable(`marché_test`), alors

- Si eff(`marché_test`) > eff(`meilleur`), alors `meilleur` $\leftarrow$ `marché_test`

- sinon, on a bien que `meilleur` est plus efficace que `marché_test`

Sinon, si $num\_token(marche\_test) > m$, alors on ne fait pas d'appels récursifs, et étant donné que les appels nous ont menés jusqu'au marché `marché_test`, cela veut dire que les marchés précédents ne permettent pas d'acheter `a_acheter`, et donc `meilleur` est toujours maximum.

Induction :

Notre appel récursif se fait sous cette forme :

Listing 18: Appels récursifs du rendu de monnaie

```
pour (indice de prochain_indice a n)
{
    nouveau_marche = cree_prochain_marche(marche_test, indice)

    appel_recursif(nouveau_marche)
}
```

On a clairement la terminaison de la boucle pour.

On pose l'invariant I comme suivant :

"si est_utilisable(meilleur), alors $\forall i \in \{prochain, ..., k\}, \forall$ marché découlant de $nouveau\_marche_i$, $eff(meilleur) \geq eff(nouveau\_marche_i)$. Sinon, $meilleur = \emptyset$"

Première itération

Au début de l'itération, d'après PRECOND, si non est_utilisable(meilleur), Alors $meilleur = \emptyset$.

Sinon, $k = 0$, donc on a bien l'inégalité de I.

Finalement, I est vraie à l'initialisation.

Récurrence

Soit k un entier entre $prochain$ et n, on suppose que notre invariant est vrai pour l'itération k.

Sachant qu'on a supposé que POSTCOND de l'appel récursif sur $nouveau\_marche_k$ est vrai, on obtient que $\forall marche$ découlant de $nouveau\_marche_{k+1}$, $eff(meilleur) \geq eff(marche)$.

Or, par hypothèse de récurrence, on avait que meilleur était déjà le maximum des marchés découlant des k premiers nouveau_marchés, il est donc maintenant le meilleur des marchés découlant des $k + 1$ nouveaux_marchés.

On a donc par récurrence : "si est_utilisable(meilleur), alors $\forall i \in \{prochain, ..., n\}, \forall$ marché découlant de $nouveau\_marche_i$, $eff(meilleur) \geq eff(nouveau\_marche_i)$. Sinon, $meilleur = \emptyset$"

Ce qui est équivalent à POSTCOND, et qui montre la correction de l'algorithme.

3 Comparaison de deux marchés

Afin de comparer deux marchés, nous utilisons une fonction `eff(marché)` qui associe à un marche le flottant suivant :

$$\sum_{jeton \in marche} \frac{n\_ressources(a\_acheter \cap jeton)}{n\_ressources(a\_acheter)}$$

Ainsi, on a $eff(marche) < 1$ si on essaye de payer avec des jetons qui ne servent pas à l'achat, et on a $eff(marche) > 1$ si on utilise plus de jetons servant à l'achat qu'il ne le faut.

Afin de comparer deux marchés, on considère donc la relation d'ordre suivante :

$$marche1 < marche2 \Leftrightarrow |1 - eff(marche1)| < |1 - eff(marche2)|$$

Cette relation d'ordre nous permet de discriminer les marchés ayant une efficacité proche de 1.

Remarque 5

Si deux marchés ont la même efficacité, on considère que celui qui est inférieur est celui qui utilise le moins de jetons.

VI Difficultés de mise en oeuvre

1 Implémentation des pouvoirs

Il y a plusieurs manières de mettre en place les pouvoirs, qui ont chacune des avantages et des inconvénients.

Implémentation avec un switch

Comme présenté précédemment (cf. 4.1), nous avons décidé de les implémenter à l'aide de pointeurs de fonctions, comme proposé dans le sujet. Cependant, une implémentation possible des pouvoirs aurait pu être l'utilisation du type `enum skill_id` et utiliser un `switch` dans la fonction exécutant le tour d'un joueur. Ainsi la fonction `skill_exec` ressemblerait à quelque chose comme suivant :

Listing 19: Pseudocode de la version possible de skill_exec avec un switch

```
enum skill_ids {SKILL_1, SKILL_2, ...}

skill_exec(skill_id)
{
    switch (skill_id)
    {
        case SKILL_1:
            /* Execute SKILL_1 */

        case SKILL_2:
            /* Execute SKILL_2 */

        ...
    }
}
```

Cela aurait eu comme avantage le fait de pouvoir avoir une exécution personnalisée de chaque pouvoirs. Ainsi, si un pouvoir nécessite de demander au joueur un des informations supplémentaires, comme la cible du pouvoir, cela est faisable.

Le problème de cette implémentation, c'est que l'ajout de nouveaux pouvoirs est fastidieux, et nécessite la modification de la fonction exécutant les pouvoirs `skill_exec`, on se retrouve pour un pouvoir donné avec une division de la localité de la logique de l'exécution de ce dernier, ce qui n'est pas idéal pour la maintenabilité du code.

Implémentation avec des pointeurs de fonctions

De l'autre côté, l'utilisation de pointeurs de fonctions permet de facilement créer de nouveaux pouvoirs et de les ajouter dans le jeu. On a juste à ajouter le nouveau pouvoir dans le tableau les contenant tous, et à ajouter un identifiant pour ce pouvoir, car la fonction `skill_exec` se réduit globalement à ce quit suit :

Listing 20: Pseudocode de la version de skill_exec avec des pointeurs de fonction

```
enum skill_id {SKILL_1, SKILL_2, ...}

skill_exec(skill_id)
{
    skills[skill_id](args);
}
```

La contrainte que cela impose est que tous les pouvoirs doivent avoir la même signature. Ainsi, si certains pouvoirs doivent avoir accès à plus d'informations que d'autres, la signature des pouvoirs doit le permettre.

Pour créer une telle signature, encore une fois, plusieurs options sont possible. La première serait de dire qu'on passe en paramètre une union de types, avec chaque type qui serait associé à un pouvoir, décrivant les données nécessaire à son exécution. Même si ça semble bien fonctionner sur le papier, pour créer l'argument à passer en paramètre, il faut passer par un `switch` ou quelque chose de similaire nous ramenant au cas de l'implémentation précédente.

Nous avons donc au final opté pour passer en paramètre 2 arguments, le tour courant et ce qui a déclenché le pouvoir. Cela restreint légèrement les possibilités des pouvoirs, et nous avons du nous résoudre à limiter certains d'entre eux, ils ont pour certains un comportement aléatoire, c'est-à-dire que lorsqu'il est question de voler un jeton, on ne laisse pas le choix au joueur et on en prend un au hasard.

Dans le cadre de nos joueurs ayant un comportement de toute façon aléatoire, ceci ne pose pas problème, mais il est normal de vouloir que le joueur puisse avoir le choix, surtout si on décidait d'intégrer des stratégies à nos joueurs ou de faire jouer des humains.

Amélioration possible

Ainsi, nous avons imaginé, sans avoir pris le temps de l'implémenter, qu'associer à chaque pouvoir un "pre-pouvoirs", une fonction se chargeant de demander aux joueurs les arguments nécessaire au bon fonctionnement du pouvoir. Ainsi, en ayant la fonction `execute_skill`, n'aurait qu'a exécuter le "pre-pouvoir" puis le pouvoir associé à `skill_id`. Cette implémentation permettrait, pour ajouter un pouvoir, de n'avoir qu'à ajouter la fonction pouvoir et la fonction "pre-pouvoir" et à les rajouter dans un tableau.

Listing 21: Pseudocode de la version de skill_exec avec les "pre-pouvoirs"

```
enum skills_id {SKILL_1, SKILL_2, ...}
```

```
skill_exec(skill_id, args)
{
    void* custom_args = preskill[skill_id](args);
    skills[skill_id](args, custom_args);
}
```

2 Stockage des architectes et des jetons

Problème

Les architectes, les jetons et les associations de pouvoirs sont générés puis stockés en statique dans `builder.c`, `token.c` et `skills.c`, durant le reste de la partie on utilise seulement les pointeurs et les méthodes de ces derniers. L'avantage est que l'on a pas besoin de connaître l'implémentation de la structure pour pouvoir interagir avec.

Cela pose un problème, ne connaissant pas l'architecture de ces structures en dehors de où elles sont créées, il nous est impossible de générer plusieurs decks en parallèle ou de les stocker ailleurs. Dans notre cas de figure cela signifie qu'on ne peut stocker les jetons et les architectes dans notre structure de partie. On ne peut donc pas jouer 2 parties, puis afficher la première.

Solutions envisagées

La solution la plus simple serait de rendre public la structure d'architecte pour pouvoir les stocker la structure de partie. Mais cela nous est impossible à cause des caractéristiques du sujet.

La solution retenue est de devoir régénérer les architectes, les jetons et les pouvoirs à partir des paramètres (graines) de la partie avant d'afficher ou de modifier une partie différente.

VII Évaluation de parties

Problématique

Maintenant que nous sommes capable de jouer des parties de manières indépendantes, il est intéressant de trouver quel couple de graine donne lieu aux parties les plus viables. Pour cela nous avons besoin d'avoir accès aux statistiques de la partie et de jouer un grand nombre de parties avec des graines différentes.

1 Extraction des statistiques d'une partie

Lorsque qu'un tour est joué, on retourne les statistiques de ce dernier à travers la structure `turn_statistics` qui est ensuite ajouté aux statisques globales de la partie au travers de la structure `game_statistics`.

Listing 22: Structures pour récupérer les statistiques

```
struct turn_statistics
```

```
{
        enum choice choice;
        int used_favor;
        int used_skill;
        int num_picked_tokens;
        int forced_skip;
};


struct game_statistics
{
        int choices[NUM_CHOICE];
        int used_favor;
        int used_skill;
        int num_picked_tokens;
        int forced_skip;
        int nb_turns;
        int result;
};
```

2 Test d'un grand nombre de parties

Maintenant que l'on peut récupérer les statistiques d'une partie, on teste chaque couple de graines avec 100 graines aléatoires (`random_seed`) pour récupérer une moyenne des statistiques pour chaque couple que l'on affiche dans la sortie standard sous la forme d'un csv. On peut alors récupérer ces données dans un fichier pour les analyser.

Listing 23: Affichage des résultats

```
seed_builders;seed_token;choices;used_favor;used_skill;
   num_picked_tokens;forced_skip;nb_turns;result
1;0;1.85,6.26,1.12;1.27;1.58;12.37;0.01;9.23;0.48
2;0;1.08,7.33,1.10;0.99;1.76;14.23;0.13;9.51;0.35
3;0;1.39,6.66,1.08;0.98;1.22;13.16;0.07;9.13;0.42
4;0;1.47,6.60,1.05;0.98;1.36;13.19;0.05;9.12;0.43
5;0;1.32,6.29,0.96;0.92;1.85;12.11;0.03;8.57;0.53
6;0;1.48,6.74,1.08;0.96;1.04;13.22;0.04;9.30;0.35
7;0;1.84,6.11,1.01;0.97;2.57;12.06;0.02;8.96;0.60
8;0;1.14,6.62,1.06;0.97;1.60;13.03;0.03;8.82;0.55
9;0;1.67,6.57,1.01;0.96;1.23;13.03;0.04;9.25;0.43
...
```

3 Analyse des résultats

A l'aide d'un programme écrit en `Python`, on peut visualiser l'influence des différentes graines sur différents paramètres.

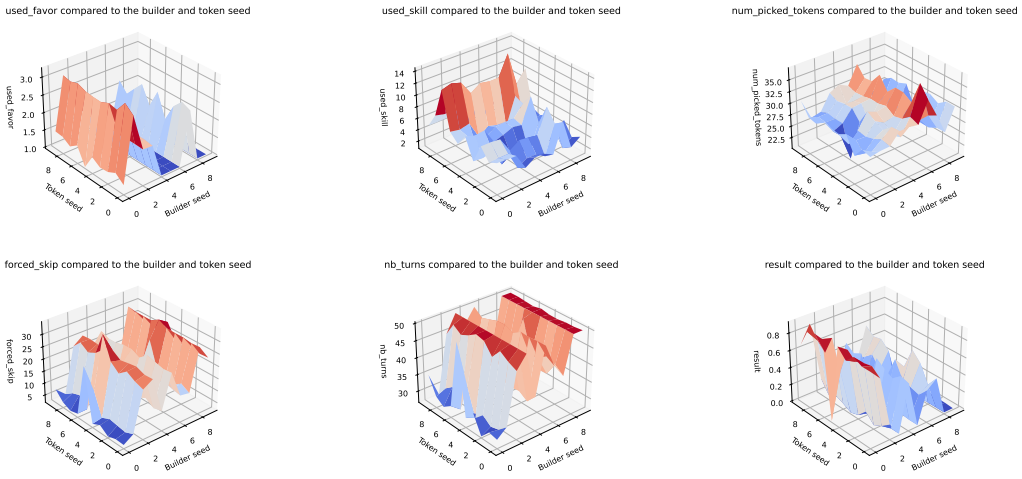


Figure 12: Influence des graines sur différents paramètres de la partie

Remarque 6

Précision : la graine 0 n'est pas générée aléatoirement mais correspond à un deck généré à la main afin d'avoir un jeu équilibré (selon nous).

A partir de ces graphiques, on remarque un comportement intéressant : la graine des architectes a beaucoup plus d'influence sur la viabilité de la partie (beaucoup de tour et pas trop de tour passés) que celle des jetons. Par exemple on voit que les joueurs sont forcés à passer souvent leur tour avec la graine d'architecte n°3 car il y a seulement 2 architectes générés dans cette dernière.

VIII Interface graphique

Problématique

Après avoir travaillé sur le fonctionnement général du jeu, il est intéressant de développer une nouvelle manière de visualiser une partie. On veut notamment pouvoir naviguer entre les tours, avoir un maximum d'information sur la partie. On s'impose également de développer cette interface en C standard.

Remarque 7

La conception flexible du code permet d'intégrer facilement une interface graphique au jeu. Cela offre une manière plus simple de visualiser une partie.

1 Stratégie

Grâce à la structure de partie (cf 5), on a accès à l'ensemble des tours d'une partie donnée. On a donc besoin de savoir afficher un tour en entier. Pour cela on scinde l'écran en plusieurs parties.

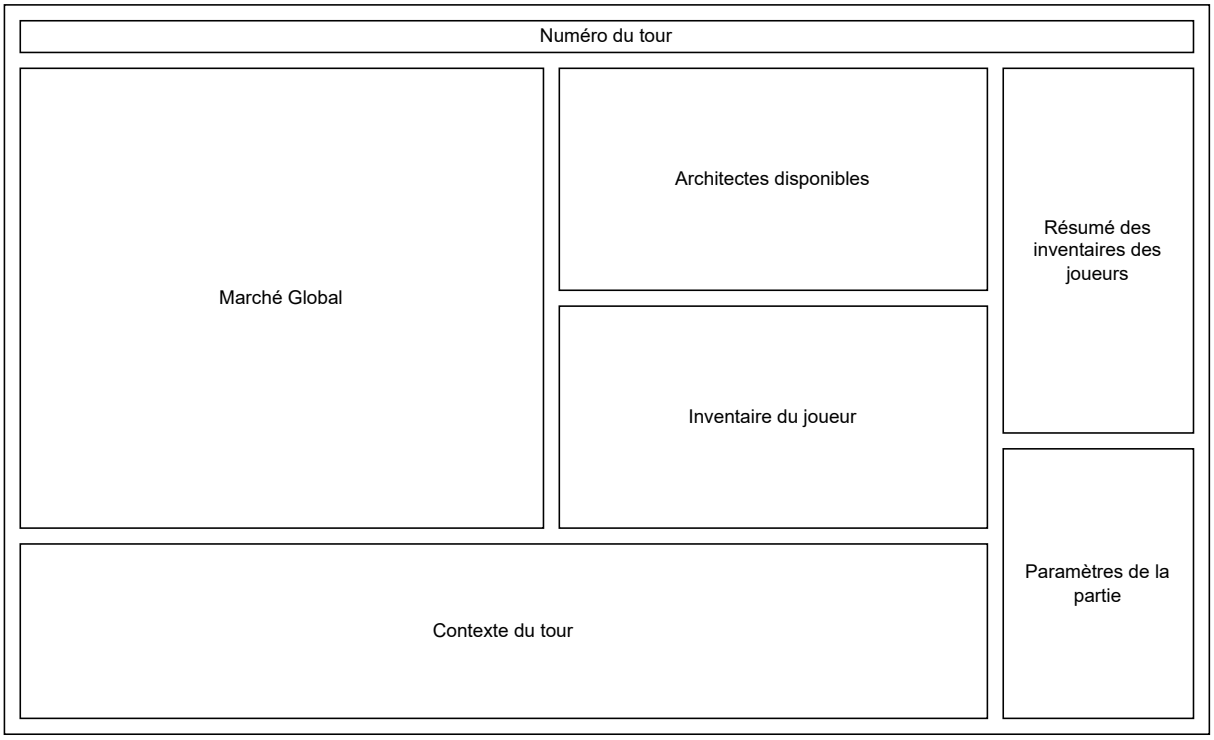


Figure 13: Schéma de l'interface

2 Mise en oeuvre

A l'aide de la fonction `print_to_coordinates` on est capable d'écrire aux coordonnées (x,y) le texte que l'on souhaite.

Par la suite, on doit réécrire toutes les fonctions d'affichage afin de pouvoir afficher les composants aux coordonnées souhaitées. On réutilise la même arborescence du code que pour le code principal. Un nouvel executable `cli` est alors créé (comme pour les tests IV et l'évaluation de parties VII) pour jouer une partie et l'afficher.

Pour naviguer entre les tours, on récupère l'entrée de l'utilisateur à l'aide de `getchar` et on décide quel tour on souhaite afficher. On peut naviguer avec les touches 'n' et 'p' ou bien les flèches.

Listing 24: Navigation dans les tours de la partie

```
while (ch != 'q')
{
    switch (ch)
    {
        /* get the right turn to display */
    }

    /* display the turn */
    cli_turn_display(turn);

    ch = getch();
}
```

Remarque 8

Lorsque qu'on essaye d'aller au delà du dernier tour, on affiche la page des résultats avec un feu d'artifice.

Conclusion

Ce projet nous a été bénéfique en termes de travail collectif et de répartition des tâches. Nous avons dû nous imposer des pratiques afin de rendre plus simple la relecture et la réédition du code.

La mise en œuvre rigoureuse de principes tels que la séparation judicieuse des fichiers, l'adoption de conventions de codage claires, la gestion de la compilation avec l'utilisation de l'outil `make`, et l'intégration de la méthode Kanban ont contribué à créer un environnement de développement plus propice à l'efficacité et à la qualité du travail produit.

Les compétences acquises seront précieuses pour les futurs projets à venir. Ce projet a donc été une vrai source de progression personnelle aussi bien du point de vue de la programmation que de l'organisation.