Rapport Projet S5-17132
Mansuba

Durand Arthur /—/ Hamouche Luxel

Responsable de projet: Renault David
Enseignant référent: Orhan Diane

Janvier 2023

Contents

Introduction

Notre projet se base sur l'exemple d'un mansuba, considéré comme l'ancêtre Perse des échecs dont le principe majeur est de mettre l'adversaire mat. Le but est donc d'implémenter tout un systeme de jeu et de relations afin de produire un jeu de plateau dans l'esprit du mansuba ou des échecs.

1.1 Présentation du projet

1.1.1 Règles

Le jeu est soumis à certaines règle afin d'encadrer sa création. Pour mieux comprendre les différentes implémentations, voici les principales :

† Le monde: Il est représnté par un ensemble de positions. Il est également définit par sa longueur et sa largeur qui en les mutltipliants nous donne le nombre maximum de posistions.

† Les relations : C'est se qui relie les positions entre elles en fonctions des directions. On peut de ce fait définir **les voisins** comme toutes les positions qui sont reliées entre elles. On a donc un maximum de voisins en fonction des déplacements autorisés.

† Déplacement simple: Il est définit comme étant le changement de position pour une position voisine disponible.

† Saut simple : Il est possible si la position voisine n'est pas disponible de sauter par dessus l'obstacle. Cependant si la position derrière est occuper ou que la structure du monde ne le permet le saut est impossible.

† Victoire : Elle définit l'arret du jeu, soit car le nombre de tour max est atteint, soit car un joueur a réussit avec une pièce à atteindre les positions de départ de l'autre joueur.

1.1.2 Stratégie de résolution

Notre stratégie pour effectuer ce projet est basé sur les tests ainsi que sur la modularité.
En effet tout au long du projet nous avons fait en sorte que toutes les valeurs utilisées puissent varier en fonctions des demandes de l'utilisateur. On a donc utilisés le moins de constantes possible pour pouvoir répondre à des changements basique sans rajouter ou changer tout une partie du projet.
Egalement notre méthode pour implémenter de nouvelles fonctionalités est dites par les tests. Pour cela on fait une liste d'objectif de la fonctionalité, chaque objectif est testé avant de passer au suivant pour être sûr que la fonctionalité marche correctement une fois implémenté.

Environement et jeu

2.1 Représentation du monde

2.1.1 Géométrie du monde

La premiere chose à faire était de définir des constantes, stuctures et variables qui allait nous servir tout au long du projet tel que :

Les directions :

NORTH	SOUTH
NEAST	SWEST
NWEST	SWEST
WEST	EAST

Le type de couleur de la case :

NO-COLOR WHITE BLACK

L'occupant de la case :

NO-SORT SIMPLE-PAWN

Cette catégorie est voué à s'agrandir au vu de l'ajout de différentes sorte de pions par la suite.

Nous avons également eu besoin des constantes pour définir le monde :

- Taille: WORLD-SIZE,
- Longueur: WIDTH,
- Largeur: HEIGHT,
- Nombre de maximum de cases: UNIT-MAX,
- Nombre de joueurs: NB-PLAYERS,

Avec ces variables, nous avons créés la structure "world" qui nous à permis d'assigner à chaque case du monde une couleur et un état d'occupation, ce qui vas grandement nous servire dans la suite du projet.

2.1.2 Plateau de jeu

Il faut definir un plateau de jeu pour l'utilisateur. Nous avons decidé de faire celui-ci de forme torique afin rendre le jeu plus modulable. Il a donc fallut nous affranchire des effets de bord à l'aide de modulos. Ce choix a aussi été fait pour des raisons d'antissipation des futures modifications, qui seront plus faciles a implémenter en partant d'un modèle très généraliste tel que le tore (Figure **??**).

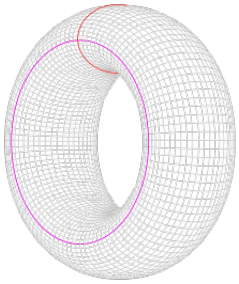
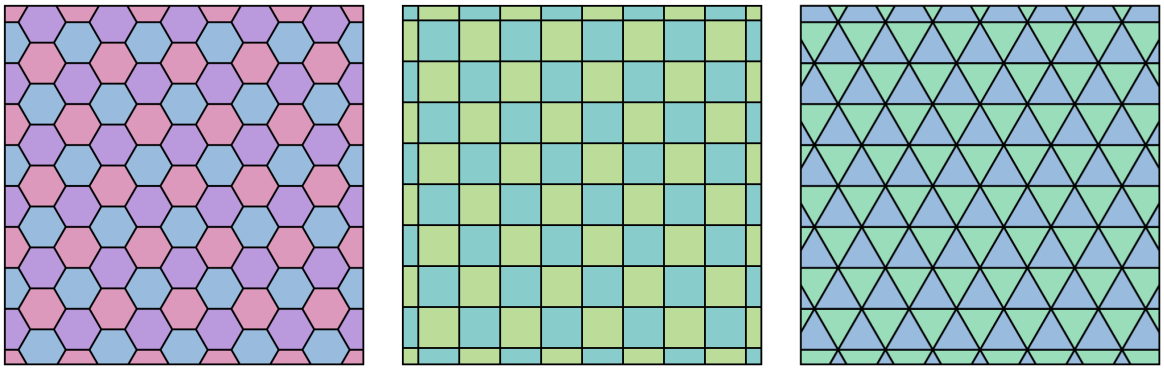


Figure 2.1: tore de jeu

Par défault, nous avons implémentés des relations entre cases de type hexagonal (Figure **??**), comme il nous l'était imposé en début de projet. Cependant, dans le cadre d'un achievement nous avons également implémenté des changement de terrains, qui impactent les relations faisant passer notre monde d'un pavage hexagonal, à un pavage carré (Figure **??**) ou triagulaire (Figure **??**). Ces changements interviennent au bout d'un certain nombre de tour, nombre qui peut être paramétré par le joueur.



(a) Pavage hexagonal (b) Pavage carré (c) Pavage triangulaire

Figure 2.2: Les différents pavages disponibles.

2.2 Pièces et joueurs

2.2.1 Pièces

Premièrement, nous avons choisis d'implémenter de simple pions, pouvant se déplacer d'une seul case (seulement si celle-ci est vide), pour des raisons de facilité principalement. Par la suite, nous avons incorporés au jeu de nouvelles pieces, le rendant plus interessant. Chaque pièce à un déplacement particulier.
C'est pour cela que nous avons créé une structure qui rend les pieces modulables. Chaque piece appartenant à cette structure est définit par un certain nombre d'arguments.

```
1 /** A struct representing a piece */
2 struct pawns_t {
3     int player_index;        // Numéro du joueur propriétaire du pion
4     int max_dep;             // Nombres maximum de déplacements du pion
5     enum color_t color;      // Couleur du pion
6     enum sort_t type;        // Type du pion
7     int position;            // Position du pion
8     int captured;            // Etat du pion
9 };
```

Nous avons implémenté plusieurs types de pions :

♟ **Le Pion simpe :**
Le pion est la pièce basique du jeu, il possède un seul mouvement dans la direction de son choix. Cependant il est possible d'en faire une dame d'echec en augmentant simplement son nombre de mouvement maximum afin qu'il puisse se déplacer sur de plus grandes distances.

La Tour :
Elle peut, comme aux échecs, se déplacer seulement en direction des points cardinaux. Nous avons décidé de limiter ses déplacement au Max(longueur du plateau, largeur du plateau). En effet, sans cette condition, étant donné l'apparence torique de notre plateau, ses movements pourrait être infinis.

L'éléphant :
Il se déplace uniquement suivant les 4 directions cardinales, dans le cas de base, il dispose de deux déplacement succésifs, mais cette valeur peux être modifiée.

Le Roi premier :
Il se téléporte directement sur une case portant un numéro premier, si celle-ci n'est pas occupé par un autre pion.

2.2.2 Joueurs

Un minimum de deux joueurs est nécessaire pour lancer une partie. Chaque joueur est associé à un index unique. Ils possèdent aussi une couleur, un nombre de pièces ainsi qu'un tableau qui les contients. La structure se présente comme ceci :

```
1 /** A struct representing a player */
2 struct players_t {
3     int index;                          // Numéro du joueur
4     int pawns_nb;                       // Nombre de pièces
5     struct pawns_t pawns[WORLD_SIZE/2]; // Tableau des pièces du joueur
6     enum color_t color;                 // Couleur du joueur
7 };
```

Le jeu peut également être joué à plus de deux joueurs en fonction de la taille du monde. L'implémentation de joueur tient à respecter l'équilibre du jeu. Donc si le plateau ne permet pas de répartir un certain nombre de joueurs différents à l'initialisation alors cet équilibre est romput.

2.2.3 Formations de départ

Pour plus de diversité, nous avons mis en place deux formations de départs différentes. Une premiere, classique, resemblant aux echecs (mais toujours en prennant en considération la forme torique de notre plateau).

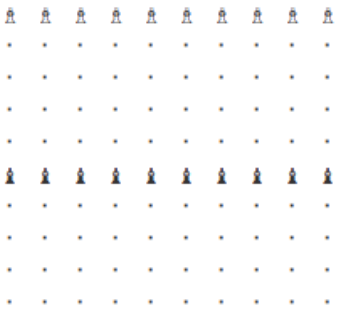


Figure 2.3: Départ classique

La deuxième formation que nous avons implémentée est plus particulière et présente le plateau sous forme de champs de bataille, avec des énorme blocs de pièces séparés par des tranchés. Cette formation avait à l'origine pour but de tester la réaction des pièces lorsqu'elles sont entourées de plein d'autres.

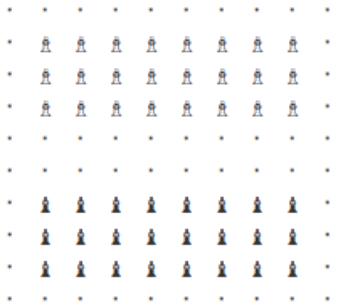


Figure 2.4: Départ champ de bataille

Le type de pions qui composent ces formations est modulable. Par défaut, la formation est remplie avec des pions simples. Néanmoins il est possible de changer cette pièce par défaut, ou encore d'ajouter un certain nombre de pièces "spéciales" en plus de la pièce par défaut. La position de ces pièces est choisie par le programme. Pour celà nous avons implémenté un algorithme qui, en fonction du nombre de pièces spéciales à placer, choisis une formation qui soit identique pour chaque joueur et qui ne crée pas de déséquilibre entre les joueurs, et ceci quelque soit le nombre de joueur.
La figure **??** montre les positions de départ pour une partie lancée avec 4 joueurs, sur un monde de taille 10×10, avec 2 tours (pièce) par joueurs.

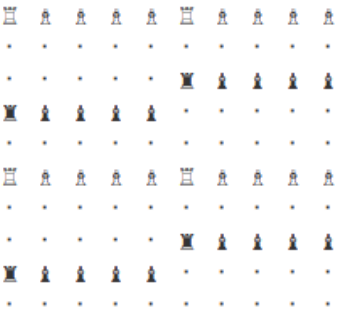


Figure 2.5: Départ classique à 4

2.3 Boucle de jeu

2.3.1 Sélection des options

Toujours dans le but d'augmenter la modularité de notre jeu, nous avons rendu un maximum d'options au choix de joueur lors du l'exécution. Les choix se font par l'intermédiaire de paramètres à ajouter lors de l'exécution du programme. Si les options sont écrites de manières incorrectes, le programme ne les prendra pas en compte et utilisera des valeurs par défaut.

2.3.2 Déroulement d'une partie

Lorsque le programme est lancé, le programme initalise un "monde extérieur", que nous verons plus tard et qui contient le plateau de jeu, les joueurs et leur pièces.

Au début de chaque tour un controle sur le changement de térrain est effectué afin de le modifié si il est nécéssaire d'après les options de l'utilisateur.
Ensuite une pièce du joueur est choisit au hasard pour se déplace dans l'une des cases accessible aléatoirement.
Puis on vérifie si il y a un vainquer ou si le max de tours à été atteint. Dans ce cas précis le jeu s'arrete et l'indique à l'utilisateur sinon on passe au tour suivant.

2.3.3 Conditions de victoire

Notre jeu admet deux conditions de victoire différente :

- La première signe la fin du jeu dès lors qu'une piece arrive dans les positions de départs de l'adversaire. Cette condition est 'simple' afin de véifier les déplacements de nos pieces et le bon fonctionnement du jeu.

- La seconde elle nécésite que la totalité des pieces arrivent dans les positions de départ de ou des adversaires. Celle-ci est beaucoup plus complèxe et nécéssite des déplacements guidé pour l'atteindre. Cependant elle permet de faire durer plus longtemps les arties notament en mode 'Champ de bataille' afin de voir beaucoup d'intéractions.

Architecture du projet

3.1 Relations

3.1.1 Voisinages

Le voisinage d'un pièce correspond aux places du monde qui sont dirrectement en contact avec la place de la pièce. Le nombre de voisins d'une pièce dépend donc de la géométrie de notre plateau (cf. **??**). On va déterminer ces derniers en testant l'entièreter des directions disponibles pour la pièce en omettant pas de verifier si les cases sont occupées ou non. Cela nous donne donc un ensemble de case disponible pour cette dernière qui repond bien aux contrainte géométrique du plateau.

3.1.2 Déplacements

Nos déplacements sont réalisés en modifiant dans le monde l'état de la position ainsi que l'index associé à la pièce en question. Cela informe donc aux autres pièces que cette position est prise et permets de savoir à quel joueur elle appartient.
Pour determiner la position exacte du déplacement, on utilise un fonctionement de recherche de proche en proche, qui va déterminer les voisins direct, puis en fonction du nombre de mouvements de la pièce, rééfectuer ce processur avec le voisin direct situé de la direction souhaitée. Le déplacement de cette dernière se fait ensuite aléatoirement parmis ces possibilités et comme dit précédement, il est impossible pour cette dernière de choisir une case déja occupée.
Dans le cas ou toutes les cases sont occupées le joueur ne peut pas jouer et sont tour est passé.

3.1.3 Chagements de terrain

Dès lors que la partie commence, une initialisation de la *seed* du terrain est nécessaire.
Cette *seed* influe directement sur les intéractions entre les différente position du monde et change donc les déplacements possibles. Chaque pièce sera donc restraintes à un certain nombre de relations :

- Pour un terrain classique (hexagonal), toutes les direction sont autorisées.
- Pour un terrain à pavage triangulaire, une case sur deux possède les même relations de voisinage, on a ici les relations (N,SE,SW) pour une partie des pièces et (S,NE,NW) pour l'autre.
- Pour un terrain a pavage carré, les seuls directions autorisés sont celle des points cardinaux (N,S,E,W).

Après avoir réduit le nombre de directions possibles, le système de déplacement est appliqué de manière identique, elle prend alors compte des contraintes de directions.
Nous avons donc réussis à implémenter un système qui en modifiant une *seed* de terrain, modifie les relations de voisinage de la partie en cours.

3.1.4 Positions de départ

En prenant en considération l'aspect torique de notre plateau nous avons souhaité le moins de déséquilibre possible au début de la partie.
C'est pourquoi La position des pièces est choisie par le programme. Pour celà nous avons implémenté

un algorithme qui, en fonction du nombre de pièces spéciales à placer, choisis une formation qui soit identique pour chaque joueur et qui ne crée pas de déséquilibre entre les joueurs, et ceci quelque soit le nombre de joueur.
La figure **??** montre les positions de départ pour une partie lancée avec 4 joueurs, sur un monde de taille 10×10, avec 2 tours (pièce, cf. **??**) par joueurs.

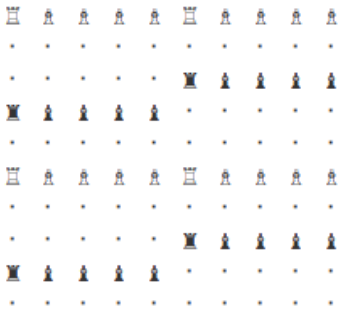


Figure 3.1: Départ classique à 4

Cette facon de penser le départ du jeu nous a permis de rendre le jeu encore plus modulable et de l'adapter à beaucoup plus de configurations que ce qui nous était demandé avec des possibilité de départ presque infinie.

3.1.5 Capture et libération

Si une pièce atterit sur une pièce adverse, une capture est réalisée. Si la position de la pièce capturée est libre, elle a une probabilité d'être libérée. Cette probabilitée est fixé à `50/100` mais peut être modifié par l'utilisateur). La question de la capture d'une pièce adverse par un joueur est un des raisons qui nous ont poussé à créer un "monde extérieur" `world_ext.c` (**cf. ??**), qui sous forme d'une structre contient le plateau, les joueurs et les pions.

```
1 struct world_ext_t {
2     struct world_t* world;                          // Plateau de jeu
3     int nb_players;                                 // Nombre de joueurs.
4     struct players_t players[WORLD_SIZE];           // Liste des joueurs.
5     struct sets_t initial_sets[WORLD_SIZE];         // Ensembles de positions de départ
      des joueurs.
6     struct sets_t current_sets[WORLD_SIZE];         // Ensembles de positions prises par
      chaque joueurs.
7     int nb_captured_pawns;                          // Nombre de pièces capturées.
8     struct pawns_t* captured_pawns[WORLD_SIZE];     // Liste des pièces capturées.
9 };
```

L'implémentation de cette fonction nous a permis d'implémenter la capture de cette façon :

- Lorsqu'une pièce est capturée, sont attribut `captured` (cf. **??**) prend la valeur `1`.
- La pièce est ajoutée à la liste des pièces capturés : `captured_pawns[]`, et sa position est retirée de la liste des positions prises par le joueur : `current_sets[index_du_joueur][]`.

Procéder de cette manière nous a permis de ne pas modifier la position de la pièce capturée, ainsi, elle la garde en mémoire. Lorsque les conditions de libérations sont remplies, ont effectue le shéma inverse, et la pièce est de retour sur le plateau.

3.2 Inclusions et organisation du projet (Makefile?)

3.2.1 Organisation du monde

3.2.2 Dépendances des fichiers

Afin de rendre le projet le plus modulable possible, nous avons séparé notre projet en plusieurs fichiers `.c`, ces fichiers contienent les fonctions qui permettent au tout de fonctionner. Chacun de ces fichiers `.c` incluent un fichier `.h` du même nom. Ces fichiers d'entête contiennent les `header` de toutes les fonctions "publiques" qui ont pour but d'être utilisés par d'autre fichiers. Les inclusions ne se font jamais entre les fichier `.c`, mais uniquement avec les `.h`.

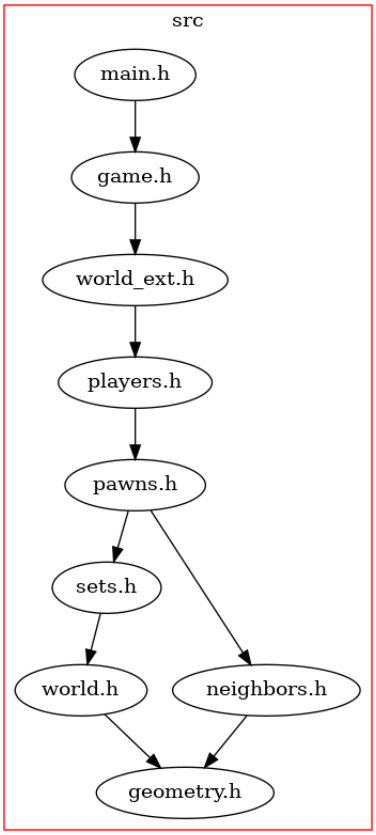


Figure 3.2: Graphique de dépendance des fichiers source. *Généré avec* `Graphviz`.

Cette organisation du projet à base d'inclusion nous permet une très grande modularité. En effet, si un fichier `.c` est modifié (comme par exemple `world.c`), le projet continue de fonctionner tant que la nouvelle implémentation respecte le fichier `.h` correspondant.

3.2.3 Compilation

Pour faciliter les travaux de compilations séparés, nous avons intégrés un fichier `Makefile`. Ce fichier nous a permis de déclarer des règles générales qui simplifient les commandes de compilations. Nous nous en somme par exemple servis pour compiler automatiquement tout les fichiers `.o` nécessaire, ou encore pour compiler et éxécuter tout les tests en même temps.

3.3 Tests

3.3.1 Structure des tests

Nous avons décidés de séparer les tests dans différentes fichier pour plus de flexibilité. Chaque fichier test contient les fonctions visants à tester un unique fichier source. Cette méthode nous à permis de pouvoir tester l'ensemble du code source en même temps, ou bien de lancer les tests d'un fichier en particulier.

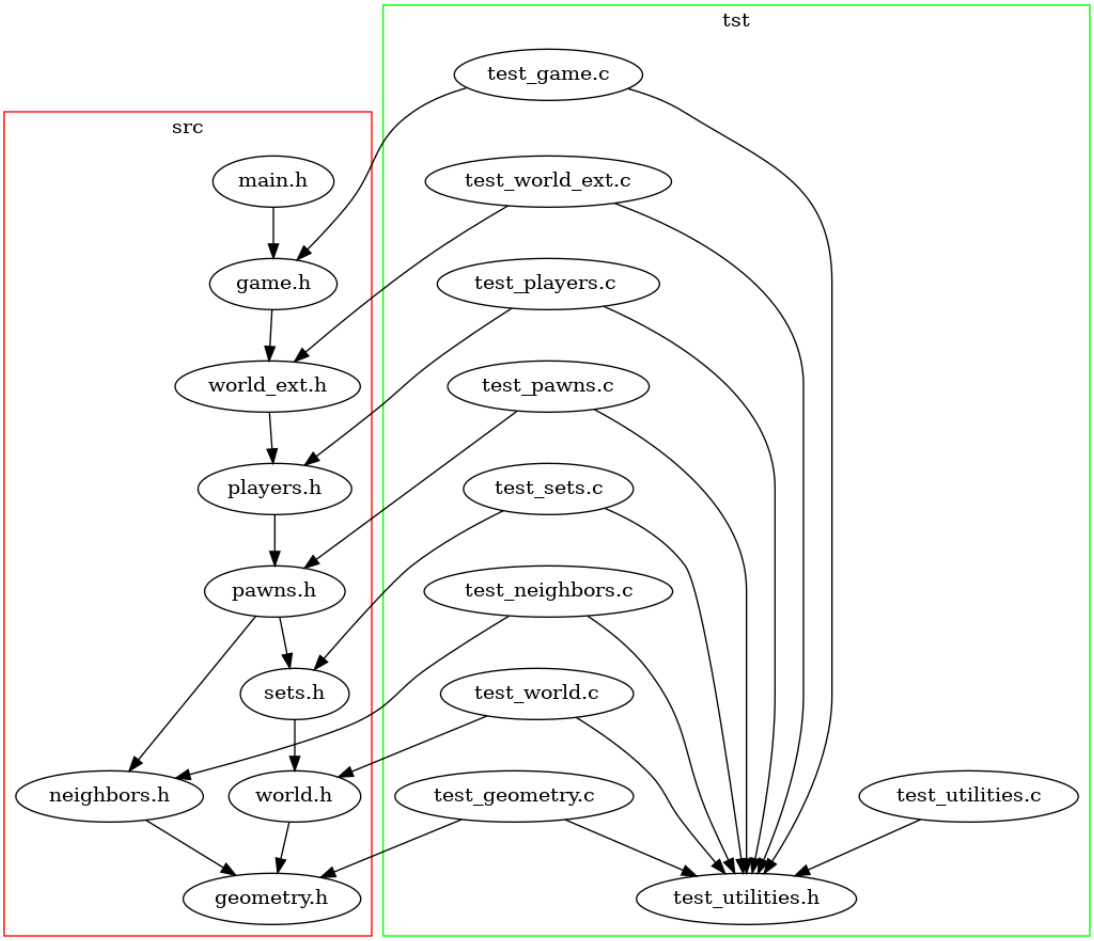


Figure 3.3: Graphique de dépendance des fichiers tests. *Généré avec* `Graphviz`.

La figure **??** montre les inclusions de nos fichiers test. On remarque que chaque fichier test inclu uniquement le fichier source qu'il teste.

De plus, nous avons décidés de séparer deux fonctions générales des tests dans un fichier `test_utilities.c`. Ce fichier est le seul des fichiers tests à avoir un fichier d'entête, car c'est le seul qui est inclu dans d'autres fichiers. Il contient les deux fonctions suivantes :

```
1 void str_test(const char str1[], const char str2[]) // Compare 2 strings
2 {
3     (!strcmp(str1, str2)) ? printf("\t\tPASSED\n") : printf("\t\tRecieve %s instead of %s
      .\n", str1, str2);
4 }
5
6 void int_test(const int int1, const int int2) // Compare 2 integers
7 {
8     (int1 == int2) ? printf("\t\tPASSED\n") : printf("\t\tRecieve %d instead of %d.\n",
      int1, int2);
9 }
```

C'est deux fonctions nous ont étés très utiles dans le cadre de la **programmation par le test**. En effet, appeller celles-ci dans nos fichier test nous a permis de très facilement comparés les retours des fonctions testés avec ce que nous attendions. En cas de réussite, le programme affiche : `PASSED`, tandis que si le test ne passe pas, le programme affichera : `Recieve <valeur_reçu> instead of <valeur_attendue>`.

Nous avons donc pu écrire nos tests, puis implémenter nos fonctions jusqu'à ce que tout nos test affiches : `PASSED`.

3.3.2 Programmation par le test

Conclusion

4.1 Difficultés rencontrées

4.2 Points à améliorer

4.3 Ce que le projet nous a apporté

- Environement de jeu (variable pions etc)
  - Structure et Variables globales:
  - Plateau :
  - Pieces :
  - Joueurs
    Les joueurs sont définis par une structure qui contient un certain nombre d'élément:
    * Un index qui diférencies les joueurs entre eux
    * Le nombre de pion qu'il possède
    * Le nombre de ses pions capturés
    * Une liste de ses pions
    * Une couleur
  - Boucle de jeu
    * Déroulement de la partie
    * Conditions de victoire
- Architecture
  - Relation de jeu
    * Relations de voisinage
    * Relation d'initialisation de partie
    * Interation entre les joueurs (prisons)
    * Interaction entre le monde et les joueurs
  - Inclusion
    * Makefile
    * Organisation des fichiers projets
    * Organisation des fichiers de test
  - Complexité
- Tests
  - Structures Tests
  - Notre utilisation des tests pour le bon focntionnement du projet
- Difficultées rencontrées
  - Problème de structure
  - Difficulté d'affichage
  - Aléatoire

Conclusion

- Points à amélioré
  - Les test
  - Travailler l'indépendance
- Ce que le projet nous a apporté
  - Découverte de Git(pas hub)
  - Découverte du Makefile
  - Progression en C