



ENSEIRB-MATMECA

1A Informatique 2020-2021

RAPPORT DE PROJET :

Les tuiles de Wang

NASDAMI QUATADAH
ROGER GAËTAN
Encadrant : **Maxime Poret**

18 décembre 2020

Table des matières

1	Introduction du sujet	3
1.1	Présentation du sujet	3
1.1.1	Achiev0 : Version de base	4
1.1.2	Achiev1	4
1.2	Problématique	4
2	Implémentation du sujet	5
2.1	Présentation des algorithmes	5
2.1.1	Présentation générale de la boucle de jeu	5
2.1.2	Implémentation des structures	6
2.1.3	Analyse des fonctions clefs	8
2.2	Difficultés rencontrées	9
2.3	Réactions face à ces difficultés	10
3	Analyse algorithmique	10
3.1	Validité des algorithmes	10
3.1.1	Les fonctions agissant sur les <i>couleurs</i>	11
3.1.2	Les fonctions agissant sur les <i>tuiles</i>	11
3.1.3	les fonctions agissant sur les <i>files</i> ou <i>maines</i> des joueurs	13
3.1.4	Le reste des fonctions	14
3.2	Complexité algorithmique	15
3.2.1	La fonction <i>list_authorized_places()</i>	15
3.2.2	La fonction <i>display_results()</i> de l'achiev1 :	16
4	Bilan	17
4.1	Rendu finale du jeu	17
4.2	Options de compilation	19
4.3	Conclusion	19

1 Introduction du sujet

1.1 Présentation du sujet

Le sujet du projet consiste à créer un jeu se basant sur les pavages de Wang. Ce pavage est constitué de tuiles carrées ayant chacun de leurs côtés associée une couleur.

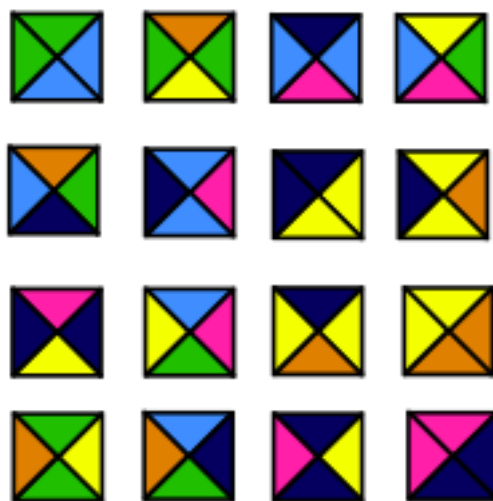


FIGURE 1 – <https://images.math.cnrs.fr>

De plus le placement de ces tuiles doit obéir a la contrainte suivante :
Si deux tuiles sont adjacentes leurs cotés en commun doivent être associés à la même couleur.

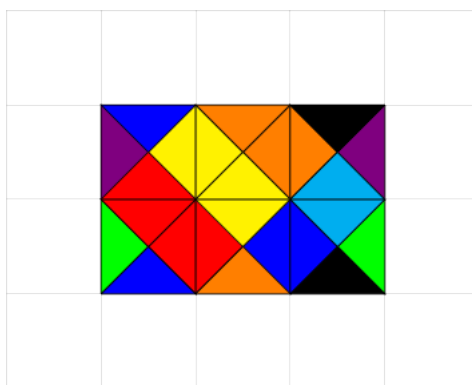


FIGURE 2 – <https://www.fl.univ-lille1.fr>

1.1.1 Achiev0 : Version de base

Dans cette version, chaque joueur possède des tuiles distribuées de manière aléatoire. Le but de chaque joueur serait de se débarrasser de toutes les tuiles qu'il possède pour gagner en les posant sur le plateau en respectant les règles citées auparavant.

1.1.2 Achiev1

Dans cette version, les joueurs cherchent à créer des motifs rapportant chacun un nombre de points, le gagnant est celui ayant le maximum de points.

exemple :

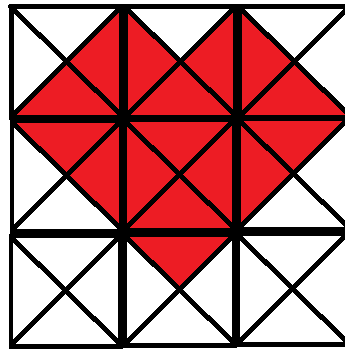


FIGURE 3 – Motif : heart

1.2 Problématique

- Implémentation des différents objets manipulés durant la boucle de jeu, à savoir les joueurs, les tuiles, le plateau .. etc.
- Analyse des différentes étapes successives nécessaires à la boucle de jeu.

2 Implémentation du sujet

2.1 Présentation des algorithmes

2.1.1 Présentation générale de la boucle de jeu

Nous pouvons représenter le programme par différentes étapes à l'aide du schéma suivant :

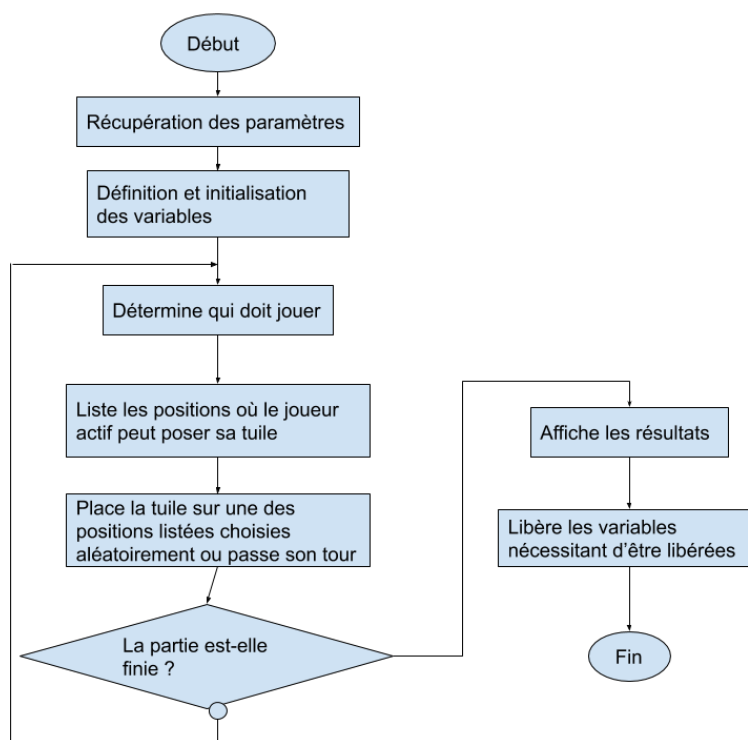


FIGURE 4 – schéma fonctionnel de la boucle de jeu

Tout d'abord le programme commence par lire et stocker les paramètres (le nombre de joueurs, la taille du plateau et la seed permettant de générer l'aléatoire). Si ces paramètres ne sont pas renseignés, le programme leur attribue des valeurs par défaut.

Ensuite les variables appelées au cours du jeu sont définies et initialisées. À partir de ce moment le jeu est initialisé, les tuiles ont été distribuées entre les joueurs et la partie peut donc commencer.

Pour ce premier tour de jeu, le premier joueur place la première tuile de sa main sur une case aléatoire du plateau. On teste ensuite si la partie est terminée, si elle ne l'est pas on passe au joueur suivant. Pour ce tour on liste

toutes les positions où le joueur peut jouer la première tuile de sa main puis il la place sur l'une de ces positions aléatoirement. Si il ne pouvait pas jouer cette tuile, il passe son tour. On teste ensuite si la partie est terminée, si elle ne l'est pas on recommence ces opérations en passant au prochain joueur.

La partie se termine lorsque un joueur a fini de poser toutes ses tuiles ou si tous les joueurs ont passé leur dernier tour.

Une fois la partie terminée, le programme calcule et affiche les scores des joueurs puis affiche ensuite le gagnant de la partie. Finalement, il libère l'espace mémoire alloué durant l'initialisation des variables.

2.1.2 Implémentation des structures

Etudions maintenant les différents objets que nous manipulons dans ce projet ainsi que la manière dont ils sont implémentés.

Une *couleur* est définie comme deux chaînes de caractères, une représentant son nom et l'autre son code.

```
1 struct color{
2     const char name[MAX_STR];
3     const char code[MAX_STR];
4 };
```

Une *tuile* est définie comme un tableau de 4 pointeurs de couleurs, un pour chaque directions (Nord,Sud,Est,Ouest). Nous utilisons des pointeurs et non directement les couleurs elles mêmes pour pouvoir changer l'implémentation des couleurs sans impacter l'implémentation des tuiles. Pour cette même raison tous les autres objets utiliseront des pointeurs de tuiles et de couleurs et non les objets eux même.

```
1 struct tile {
2     struct color* tile_colors [4];
3 };
```

Nous devons ensuite créer une structure file. Pour cela nous avons implémenter une liste simplement chaînée FIFO (First In First Out). Cette liste est implémentée de la manière suivante : On définit d'abord une structure *Element* qui contient l'adresse d'une tuile et l'adresse d'un autre *Element*(suivant). Ainsi on peut passer d'un *Element* à un autre et donc définir une chaîne d'*Element*. On définit alors la structure *Queue* qui contient l'adresse du premier *Element* de la file (le prochain *Element* qui sera supprimé de la file) et à l'aide du chaînage précédemment décrit on peut donc remonter à tout les éléments de la file. Le dernier *Element* de la file (le dernier élément ajouté à la file) renvoie vers l'adresse NULL. Ainsi on sait qu'on se trouve à la fin de la file. Ces files représenteront les mains des joueurs.

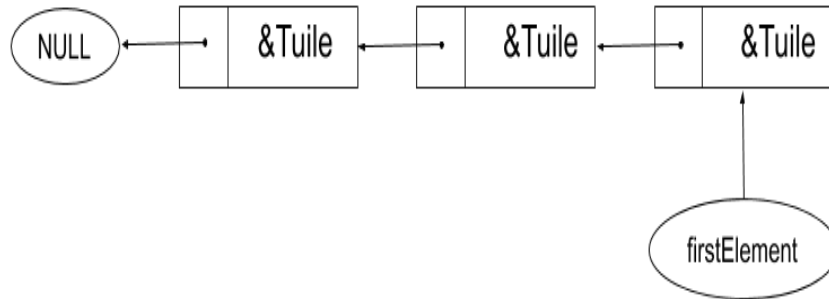


FIGURE 5 – schéma d'une file FIFO simplement chaînée

Nous avons ensuite créé une structure *board* représentant le plateau de jeu. Cette structure est composée d'une matrice d'adresse de tuiles représentant les tuiles sur le plateau. On considère que cette adresse est NULL si aucune tuile n'est posée à cette position. La structure est aussi composée d'un entier **size** qui indique la taille d'un côté du plateau (le plateau étant un carré).

```

1 struct board{
2     const struct tile* places[MAX_BOARD_SIZE][MAX_BOARD_SIZE];
3     int size;
4 };
  
```

La structure *positions* permet de représenter une liste de positions du plateau. Elle se compose d'une liste d'abscisses, une liste d'ordonnées ainsi qu'un **entier** indiquant le nombre de positions listées.

```

1 struct positions{
2     int i[MAX_BOARD_SIZE*MAX_BOARD_SIZE];
3     int j[MAX_BOARD_SIZE*MAX_BOARD_SIZE];
4     int length;
5 };
  
```

La structure *player* représente un joueur. Elle se compose de l'adresse d'une file représentant sa main, d'un entier représentant son score ainsi qu'une structure *positions* définissant les positions du plateau où le joueur à poser une de ses tuiles.

```

1 struct player{
2     Queue *cards;
3     int score;
4     struct positions playertiles;
5 };
  
```

La structure *players* représente l'ensemble des joueurs. Elle se compose d'une liste de *player* représentant les joueurs, un entier indiquant le nombre de joueurs ainsi qu'un entier indiquant le joueur actif de ce tour.

```

1 struct players{
2     struct player player[MAX_PLAYERS];
3     int length;
4     int rank;
5 };

```

La structure *motif* représente les formes que doivent créer les joueurs à l'aide d'une même couleur pour obtenir des points. Ces motif sont implémentés par une taille, le score qui lui est associé ainsi qu'une matrice de tuile définissant le motif. Cette matrice contient des tuiles composées des couleurs *noir* et *blanche*. Les zones blanches de ces tuiles représentent les zones appartenant au motif.

```

1 struct motif{
2     struct tile motiff[MAX_MOTIF][MAX_MOTIF];
3     int distance;
4     int score_motif;
5     int rank;
6 };

```

Les objets couleur, tuile, motif et file possèdent (avec les fonctions qui leurs sont associées) chacune leur propre fichier. Toutes les autres structures se trouvent dans un même fichier *player.c*

Ainsi on peut constater des dépendances entre les différents fichiers :

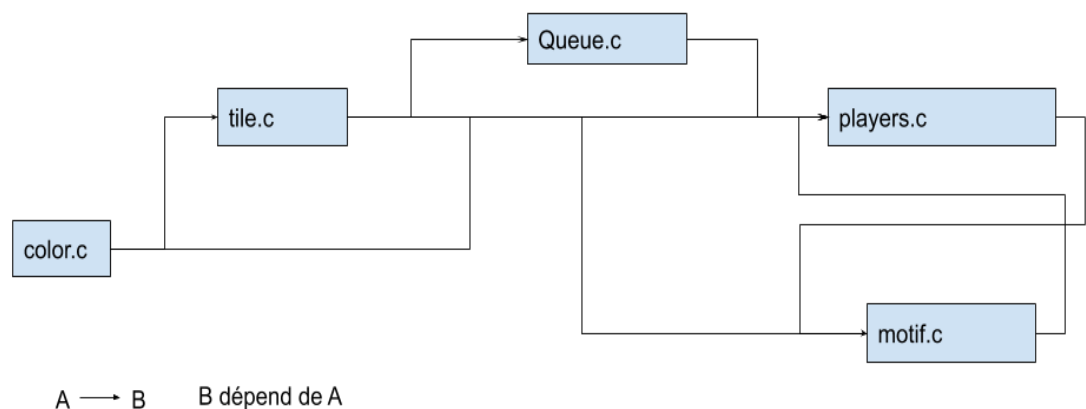


FIGURE 6 – schéma des dépendance entre les fichiers

2.1.3 Analyse des fonctions clefs

La plupart des fonctions que nous avons dû implémenter dans ce projet sont assez simples et furent facilement implémentées. Cependant, d'autres

fonctions qu'on qualifierait clefs ont nécessité plus de réflexion et donc une analyse plus approfondie.

La fonction *split_deck()* prend en argument un deck ainsi que les joueurs et doit distribuer aléatoirement les tuiles du deck entre les joueurs. Pour cela nous avons d'abord déterminé combien de tuiles chaque joueur devait obtenir. Ensuite pour chaque tuile du deck nous déterminons aléatoirement un joueur à qui donner celle-ci. Si il a déjà toutes ses tuiles on la donne au joueur suivant et ainsi de suite.

La fonction *list_authorized_positions()* prend en argument une tuile à poser ainsi que le plateau de jeu et doit retourner la liste des positions où le joueur peut poser cette tuile. Pour cela nous parcourons chaque case du plateau et nous testons si la tuile peut être posée dans cette case. Pour cela nous testons d'abord si la case est vide, puis nous vérifions que cette case est adjacente à au moins une tuile déjà posée. Enfin nous testons si mettre la tuile à cette case respecte la contrainte du pavage de Wang(2). Si cette case remplit toutes ces conditions on l'ajoute à la liste des positions autorisées. De plus si toutes les cases sont vides (le premier joueur n'a pas encore joué) toutes les positions du plateau sont ajoutées à la liste des positions.

La fonction *display_result()* prend en argument le plateau à la fin de la partie et la liste des joueurs et renvoie les scores des joueurs. Pour cela nous commençons par parcourir la liste des joueurs. Pour chaque joueur nous parcourons ensuite la liste des tuiles qu'ils ont placé sur le plateau. Pour chacune de ces tuiles nous testons si elles appartiennent à un motif. Pour tester cela nous commençons pour chaque motif par regarder si la tuile ne se trouve pas trop près d'un bord du plateau. Ensuite nous testons si la tuile et toutes les tuiles autour de celle ci forment bien le motif. Pour cela on regarde si toutes les zones devant appartenir au motif sont bien de la même couleur. Enfin si le motif est détecté alors on incrémente les points du joueurs et on passe à sa tuile suivante.

Après avoir parcouru toute la liste des joueurs de cette façon nous affichons leurs score ainsi que le(s) gagnant(s).

2.2 Difficultés rencontrées

Au cours de ce projet nous avons rencontré plusieurs obstacles techniques qui nous ont demandés d'effectuer des recherches allant au-delà de nos acquis.

Le premier de ces obstacles fut l'implémentation de la fonction *deck_init*. Cette fonction nous a posé deux problèmes. Le premier est le fait que ayant

pour contrainte de ne pas modifier `color.h`, nous n'avions pas accès à la structure de `color` ce qui rendait la création des tuiles du deck impossible. Le deuxième fut que les tuiles étaient définies comme des variables locales de `deck_init` et donc étaient inutilisables dans le reste du jeu.

Le deuxième obstacle que nous ayons rencontré fut les erreurs de segmentations. Ce fut vraiment un problème à cause du temps que nous avons dû prendre pour pouvoir identifier et corriger ces erreurs.

Enfin la dernière difficulté à laquelle nous avons été confronté fut la correction de notre codes. Même si notre code était syntaxiquement correct, nos tests n'étaient pas cohérents avec nos attentes et identifier les erreurs dans notre code fut une tâche difficile.

2.3 Réactions face à ces difficultés

Pour surmonter ces obstacles nous avons cherché des méthodes et découvert des outils utiles.

Pour régler le premier problème avec la fonction `deck_init` nous avons simplement utilisé la fonction `color_from_name` pour avoir accès aux adresses des couleurs sans avoir à en définir. Mais nous avons mis anormalement longtemps avant de comprendre l'intérêt de cette fonction. Pour le second problème de `deck_init` nous avons d'abord pensé à utiliser l'allocation dynamique pour pouvoir utiliser les tuiles définies localement dans `deck_init`. Néanmoins, nous avons été contraint de ne pas utiliser d'allocation dynamique dans `color.c` et `tile.c` nous avons donc créé un tableau en guise de variable globale possédant l'ensemble des tuiles du jeu.

Les problèmes d'erreurs de segmentations nous ont permis de nous familiariser avec les outils *gdb* et *valgrind* qui nous ont été très utiles durant le débogage.

Enfin le temps que nous avons passé sur la correction du code montre bien l'importance de tester chaque petite avancé dans le code car le manque de testes au début de notre projet nous a bien portée préjudice plus tard.

3 Analyse algorithmique

3.1 Validité des algorithmes

Dans cette partie, on va aborder la validité des algorithmes implémentés dans le projet ainsi que les test faits dans chaque fichier.

3.1.1 Les fonctions agissant sur les *couleurs*

Comme indiqué dans l'énoncé, on a bien respecter les nominations de chaque fonction dans ce fichier à savoir : **color_name**, **color_cstring**, **color_from_name**.

Après l'implémentation des tests :

```
1 printf(" ***** test pour le fichier color.c *****\n");
2 printf("Renvoie le nom de la couleur <Green> : %s \n",
   color_name(&Green));
3 printf("Renvoie le code de la couleur <Black> : %s \n",
   color_cstring(&Black));
4 printf("Renvoie le nom de la couleur <Red> appelee a partir de
   color_from_name : %s\n", color_name(color_from_name("Red")));
5
```

Chaque ligne débutant par *printf* correspond au test d'une fonction, les premiers mots correspondent à ce que devrait retourner la fonction puis après les " :", vient le résultat du test de la fonction.

Voici donc le résultat de leurs validités :

```
***** test pour le fichier color.c *****
Renvoie le nom de la couleur <Green> : Green
Renvoie le code de la couleur <Black> : \u001b[30m
Renvoie le nom de la couleur <Red> appelee a partir de color_from_name : Red
```

FIGURE 7 – rendu du test de *color.c*

3.1.2 Les fonctions agissant sur les *tuiles*

De la même manière que dans *color.c*, les fonctions de *tile.c* ont aussi été bien prédéfinies par l'énoncé. Voici l'implémentation de leurs tests :

```
1 printf("\n\n\n ***** test pour le fichier tile.c *****\n");
2 struct tile t1 = {{ &Red,&Black,&Black,&Black }};
3 struct tile t2 = {{ &White,&Red,&White,&White }};
4 printf("teste empty_tile -> doit renvoyer (0,1) : %d,%d \n",
   empty_tile() == &t1, empty_tile() == NULL);
5 printf("teste tile_is_empty -> doit renvoyer (0,1) : %d,%d\n",
   tile_is_empty(&t1), tile_is_empty(empty_tile()));
6 printf("teste tile_equals -> doit renvoyer (0,1) : %d, %d \n",
   tile_equals(&t1,&t2), tile_equals(&t1,&t1));
7 printf("teste tile_edge -> doit renvoyer le nom de la couleur
   NORD de la tuile t2 : %s\n", color_name(tile_edge(&t2,NORTH)));
8
```

Les deux premières lignes correspondent à la création de deux tuiles pour lesquelles on implémente les tests qui suivent.

Les tests sont fait de la même manière que dans *color.c*.

Voici le rendu de ces tests :

```
***** test pour le fichier tile.c *****
teste empty_tile -> doit renvoyer (0,1) : 0,1
teste tile_is_empty -> doit renvoyer (0,1) : 0,1
teste tile_equals -> doit renvoyer (0,1) : 0, 1
teste tile_edge -> doit renvoyer le nom de la couleur NORD de la tuile t2 : White
```

FIGURE 8 – rendu du test de *tile.c*

De la même manière, ce qui vient avant les deux points ':' correspond à ce que doit retourner la fonction après ces deux points.

La fonction *deck_init* est la moins évidente à tester comme le montre le code suivant :

```
1 struct deck de;
2 deck_init(&de);
3 int i = de.size;
4 int pair = 0;
5 while(i>0)
6 {
7     printf("[%s",color_name(de.cards[pair].t->tile_colors[0]));
8     for (int dir = 1; dir<4;dir++)
9     {
10         printf(",%s",color_name(tile_edge(de.cards[pair].t,dir)));
11     }
12     printf("] : %d fois \n",de.cards[pair].n);
13     i -= de.cards[pair].n;
14     pair++;
15 }
16 printf("La taille du deck est de : %d tuiles \n",de.size);
17
```

Les principales étapes du test de la fonction *deck_init* :

- Appel à la fonction *deck_init*.
- Initialisation d'une variable *i* avec la taille du *deck*.
- Affichage de toutes les couleurs de chaque tuile tout en décrémentant la variable *i* pour la terminaison de la boucle *while*.
- Affichage de la taille du *deck*.

Pour l'initialisation qu'on avait faite, en voici le résultat :

```
[Red,Red,Red,Red] : 6 fois
[Red,Black,Red,Red] : 2 fois
La taille du deck est de : 8 tuiles
```

FIGURE 9 – rendu du test de *deck_init*

3.1.3 les fonctions agissant sur les *files* ou *main*s des joueurs

Il s'agit des tests des fonctions du fichier *queue.c* qui rappelons le, manipulent les tuiles dans la main du joueur.

- La fonction *initQueue()* initialise la structure **Queue** en mettant son premier élément comme **NULL**.
- La fonction *push()* rajoute un élément à la file entrée en paramètres, puis le met dans son rang en respectant les règles d'une file.
- La fonction *top()* retourne le premier élément de la file et la fonction *pop()* se charge de le supprimer.
- La fonction *popAll()* supprime tous les éléments de la file sauf *NULL* qui après suppression devient donc le premier élément de la file.
- La fonction *rand_q()* qui sert à mélanger les tuiles avant de les distribuer.

L'implémentation des tests donne :

```
1 printf("\n\n\n ***** test pour le fichier queue.c *****\n");
2 Queue* q = initQueue();
3 printf("teste si le premier element de la file est NULL -> doit
   envoyer 1 : %d\n",q->firstElement == NULL);
4 push(q,&t1);
5 push(q,&t2);
6 printf("teste top -> doit envoyer 1 : %d\n",top(q)->tile == &t1)
   ;
7 pop(q);
8 printf("teste si t1 a bien ete supprime -> doit envoyer 1 : %d\n
   \n",top(q)->tile == &t2);
9 popAll(q);
```

Le rendu donne :

```
***** test pour le fichier queue.c *****
teste si le premier element de la file est NULL -> doit envoyer 1 : 1
teste top -> doit envoyer 1 : 1
teste si t1 a bien ete supprimee -> doit envoyer 1 : 1
```

FIGURE 10 – Rendu du test des fonctions de *queue.c*

En ce qui concerne la fonction *rand_q()* :

```

1 printf("*****test de rand_q*****\n");
2 printf("file non melangee : ");
3 for (int i =0; i<5; i++)
4 {
5     push(q,&t1);
6     printf("t1 ");
7 }
8 for (int i =0; i<5; i++)
9 {
10    push(q,&t2);
11    printf("t2 ");
12 }
13 q = rand_q(q);
14 printf("\nfile melangee : ");
15 while (top(q) != NULL)
16 {
17     const struct tile *t = top(q)->tile;
18     if (t == &t1)
19         printf("t1 ");
20     if (t == &t2)
21         printf("t2 ");
22     pop(q);
23 }

```

Ce qui donne comme rendu le résultat suivant :

```

*****test de rand_q*****
file non melangee : t1 t1 t1 t1 t1 t2 t2 t2 t2 t2
file melangee : t1 t2 t2 t2 t1 t2 t2 t1 t1 t1

```

FIGURE 11 – rendu du test de *rand_q*

3.1.4 Le reste des fonctions

Le fichier *players.c* comporte toutes les fonctions qui restent et qui sont implémentées dans la boucle du jeu. Ces dernières sont directement testées par la boucle du jeu et donc on ne les testera pas dans cette partie.

Remarque : En ce qui concerne l'**Achiev1**, on a agit en majeure partie sur la fonction *display_results()*. Le but était de rajouter les scores des motifs aux joueurs les possédant et d'afficher le score finale de chaque joueur ainsi que d'afficher le gagnant ou les gagnants de la partie.

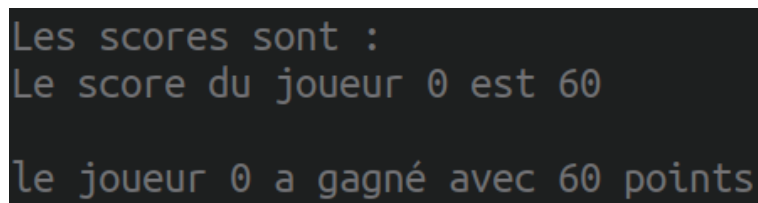
Pour tester donc cette nouvelle *display_results()*, On a limité le boardSize à

3x3, et on a laissé un seul joueur, puis on lui a distribué 9 tuiles semblables remplies de la même couleur dans toutes les directions. De plus, on sait que pour avoir le motif coeur il faut 7 tuiles de la même couleur placées sous forme d'un carré, donc la seule possibilité pour le joueur quelque soit la combinaison avec laquelle il pose ses tuiles, il finira par arriver à faire le motif coeur et seulement lui. D'où, son score finale sera bien celui du coeur à savoir 60.

Si on fait pareil pour BoardSize (4 x 4) et on distribuant 16 tuiles à l'unique joueur, il finira par arriver à faire 4 coeurs, son score finale serait donc de $240 = 60 * 4$.

```
1 // Board size
2 unsigned int boardSize = 3;
3 // number of players in the game
4 unsigned int playersNumber = 1;
```

Le résultat suivant montre bien que la fonction marche :



```
Les scores sont :
Le score du joueur 0 est 60
le joueur 0 a gagné avec 60 points
```

FIGURE 12 – rendu de la fonction *display_results*

3.2 Complexité algorithmique

Les fonctions qui viennent dans *color.c* et *tile.c* et *queue.c* sont soit de complexité constante ou linéaire. Dès lors la création de la structure *Board* qui contient une matrice de tuiles, les fonctions qui utilisent cette structure ont des complexités bien plus élevées puisqu'elles sont dans l'obligation de parcourir le plateau de jeu (qui est une matrice).

Dans cette partie on va aborder ces fonctions :

3.2.1 La fonction *list_authorized_places()*

Cette fonction a pour but, rappelons le, parcourir le plateau de jeu puis retourner une liste de positions dans *struct positions* dans lesquels il est possible pour le joueur actif de déposer une de ses tuiles. Cette fonction donc doit :

- Parcourir le plateau qui est pour nous une matrice carrée.
- Chercher où est-ce que le joueur pourrait déposer sa tuile entrée en paramètres, et donc cette étape requiert le parcours des 4 directions de sa tuile.

Ce qui donne en code :

```

1 ...
2 for (int k=0; k<b->size; k++){
3     for (int l=0; l<b->size; l++){
4         ...
5         for (int dir = 0; dir<4; dir++){
6             ...

```

La complexité donc de cette fonction est de l'ordre $C = O((b \rightarrow size)^2 \times 4)$. Il s'agit donc d'une complexité **quadratique**, chose qui était prévue puisque notre *struct board* est sous forme d'une matrice carrée.

3.2.2 La fonction *display_results()* de l'achiev1 :

Il s'agit de la fonction ayant la plus grande complexité dans notre jeu et dont nous sommes pas fières de l'avoir implémenté de cette manière. En effet, la fonction a besoin de :

- Parcourir tous les joueurs
- Parcourir les tuiles qu'ils possèdent (que les joueurs ont précédemment déposé sur le plateau)
- Parcourir la liste des motifs permettant d'obtenir des scores (2 dans notre cas, on peut la considérer constante)
- Parcourir les cases entourant la tuile centrale du joueur pour vérifier si le joueur a réussi à faire un motif.

Ce qui donne finalement :

```

1 for (int k=0 ; k < p->length; k++)
2 {
3     for (int l=0; l < p->player[k].playertiles.length ; l++ )
4     {
5         for (int m=0 ; m<2 ; m++)
6             ...
7         for (int a=x-liste_motif[m].distance; a<x+
liste_motif[m].distance +1 ;a++)
8             {
9                 for (int z=y-liste_motif[m].distance; z<y+
liste_motif[m].distance +1 ;z++)
10                    {
11                        for (int d=0; d<4; d++)
12                            ...

```



```

        if (tile_edge(&liste_motif[m].motiff[a-x+
liste_motif[m].distance][z-y+liste_motif[m].distance],d) ==
color_from_name("White"))

```

La dernière boucle est de complexité constante, la complexité de cette fonction est de l'ordre de $O(p \rightarrow length \times playertiles.length \times 2 \times 3 \times 3 \times 4 \times len)$ En effet :

Dans l'énoncé, il est mentionnée que la distance de Chebychev ne doit pas dépasser 2, les deux avants dernières boucles utilisent chacune le double de la distance de Chebychev (qui est pour notre cas égale à 1) +1, d'où le 3 dans le calcul de la complexité. Le len à la fin correspond à la complexité de la fonction *color_from_name()* qui est linéaire, puisqu'elle utilise la fonction *strcmp* dans une boucle finie. La complexité finale est donc **cubique**.

4 Bilan

4.1 Rendu finale du jeu

Dans cette partie, nous allons présenter le rendu de la boucle finale du jeu. Les étapes se déroulent de cette manière :

1. Les joueurs reçoivent chacun des cartes.
2. La partie commence et le programme entre dans la boucle du jeu
3. suivant les consignes et règles du jeu, la partie se termine
4. On affiche les résultats finaux (le score de chaque joueur ainsi que le(s) gagnant(s) de la partie)

Lors de l'implémentation de nos fonctions, nous avons pris en compte le fait de mettre *printf* partout là où on a jugé qu'il faut le mettre pour rendre un peu vivant le rendu du jeu. Voici le résultat final pour une boucle de 2 joueurs, un deck de plusieurs tuiles (>100) qu'on va pas tout montrer :

```

le joueur 1 obtient une carte
le joueur 0 obtient une carte
le joueur 1 obtient une carte
le joueur 1 obtient une carte
le joueur 1 obtient une carte
le joueur 1 obtient une carte
le joueur 0 obtient une carte
le joueur 0 obtient une carte

```

```
le joueur 0 joue...
le joueur a placé une tuile en (4,0)
le joueur possède la tuile à la position (4,0)
```

```
le joueur 1 joue...
le joueur a placé une tuile en (4,1)
le joueur possède la tuile à la position (4,1)
```

```
le joueur 0 joue...
le joueur a placé une tuile en (5,1)
le joueur possède la tuile à la position (5,1)
```

```
le joueur 1 joue...
le joueur a placé une tuile en (3,0)
le joueur possède la tuile à la position (3,0)
```

```
le joueur 0 joue...
le joueur a placé une tuile en (4,2)
le joueur possède la tuile à la position (4,2)
```

```
le joueur 0 joue...
le joueur a placé une tuile en (8,0)
le joueur possède la tuile à la position (8,0)
```

```
le joueur 1 joue...
ce tour est skip 1
```

```
le joueur 0 joue...
ce tour est skip 2
```

Les scores sont :

Le score du joueur 0 est 180

Le score du joueur 1 est 300

le joueur 1 a gagné avec 300 points

4.2 Options de compilation

Par défaut, le jeu est mis à 2 joueurs, pour un plateau de taille 10 et pour une seed égale à 0. Sauf qu'il existe des options de compilation du jeu avec lesquelles on peut changer ces données là à savoir une compilation avec `-s` pour changer la valeur de la seed, `-n` pour changer le nombre des joueurs et `-b` pour changer la taille du plateau

4.3 Conclusion

Ce projet de programmation que nous venons de réaliser nous a permis de manipuler tous les acquis du cours et même aller plus loin que cela en découvrant de nouvelles manipulations dans ce monde vaste de *l'informatique* à titre d'exemple l'utilisation de *gdb, valgrind*. De plus, nous avons fait face à de nombreux problèmes qu'on a pu finalement résoudre.