



Rapport de projet - Semestre 6 - Groupe 15528

Mai 2022

Flood-filling Games

Filière Informatique - ENSEIRB-MATMECA

Auteurs :

KHIARI Skandar
KABBOU Meryem
MEDGHALI Majid
OUALI Mohammed

Encadrants :

M.MATRICON Théo
M.RENAULT David

Table des matières

1	Introduction au jeu du Flood	3
1.1	Contexte et présentation du sujet	3
1.2	Outils de travail	3
2	Problématique	3
2.1	Création du plateau de jeu	3
2.2	Implémentation du serveur de jeu	3
2.3	Implémentation des joueurs et de leur stratégie de jeu	3
3	Cadre du travail	4
3.1	Graphe de dépendance	4
3.2	Architecture des fichiers et détails techniques	4
4	Implémentation du serveur de jeu	5
4.1	Initialisation du plateau du jeu	5
4.2	Mise à jour du plateau du jeu	6
4.3	Boucle de jeu	6
4.4	Vérification de la correction des coups	7
5	Tests du projet	7
5.1	Les tests	7
5.2	Les Affichages	8
6	Problèmes intéressants rencontrés et solutions algorithmiques	8
6.1	Conversion coo/csr	8
6.2	La génération des couleurs du plateau de jeu	8
6.3	Implémentation de min-max	8
7	Joueurs et stratégies	8
7.1	Description des joueurs	8
7.2	Algorithmme de jeu greedy	9
7.3	Algorithmme de jeu paresseux	9
7.4	Algorithmme de jeu avancé	10
7.4.1	Arbre de jeu	10
7.4.2	Principe	11
8	Discussion	12
8.1	Limites et améliorations possibles	12
8.2	Problèmes rencontrés	12
9	Conclusion	12

1 Introduction au jeu du Flood

1.1 Contexte et présentation du sujet

Ce projet s'est déroulé dans le cadre d'une évaluation pour le semestre 6. Le sujet s'articule autour des jeux de flood-filling. Le but final était d'implémenter un ensemble de clients qui essaient de recouvrir automatiquement un plateau en le moins de coups possible, en plus d'un serveur qui doit organiser la partie entre les joueurs. Ce projet a été réalisé par groupe de quatre personnes. Le langage de programmation pour ce projet était le langage C.

1.2 Outils de travail

Afin de pouvoir travailler en équipe et de se partager le code écrit par chacun, nous avons utilisé git. Cet outil nous permet par ailleurs en cas de problème avec la dernière version de revenir à la précédente. Ce rapport est écrit en L^AT_EX, et les fichiers évoqués dans ce rapport sont présents sur le dépôt. Nous nous sommes également grandement appuyés sur l'aide de débogueur comme GDB et Valgrind.

2 Problématique

L'objectif de ce projet était de réaliser une implémentation du jeu *Flood* en langage C. Pour y arriver, nous avons rencontré plusieurs problématiques :

2.1 Création du plateau de jeu

Dans ce projet, il était nécessaire d'implémenter des plateaux de différentes formes et tailles. Et pour permettre aux joueurs et aux serveurs de savoir les voisins de chaque sommet, les plateaux devraient être implémentés à l'aide de la bibliothèque gsl.

Un plateau de jeu est défini principalement par deux caractéristiques :

- la forme qui est implémentée par un graphe qui définit les voisins de chaque sommet.
- le fond qui quant à lui permet d'implémenter les couleurs des sommets du plateau de jeu selon quelques méthodes de coloration.

2.2 Implémentation du serveur de jeu

Ce serveur devait organiser une partie en fonction des options, il doit alors initialiser chacun des clients puis les faire jouer tour à tour en leurs envoyant les coups joués par leur adversaire, le serveur avait alors pour tache de vérifier la validité des coups de chacun des joueurs. De plus, ce serveur doit aussi informer les clients de la fin de la partie.

2.3 Implémentation des joueurs et de leur stratégie de jeu

Ces joueurs ou clients doivent être automatique, ils doivent jouer tout seul après avoir été initialisé. Ils doivent donc avoir leur stratégie propre et jouer des coups valides. Ils doivent posséder et adapter leur propre graphe au cours de la partie pour être en mesure de renvoyer des coups valides. Enfin, ils doivent lorsque la partie est terminée libérer le plateau de jeu et les allocations réalisées.

Afin que les joueurs puissent jouer la partie, il faut lier dynamiquement ceux-ci, soit à l'aide du Makefile, soit à l'aide des fonctions de la famille dlopen (Méthode utilisée).

3 Cadre du travail

3.1 Graphe de dépendance

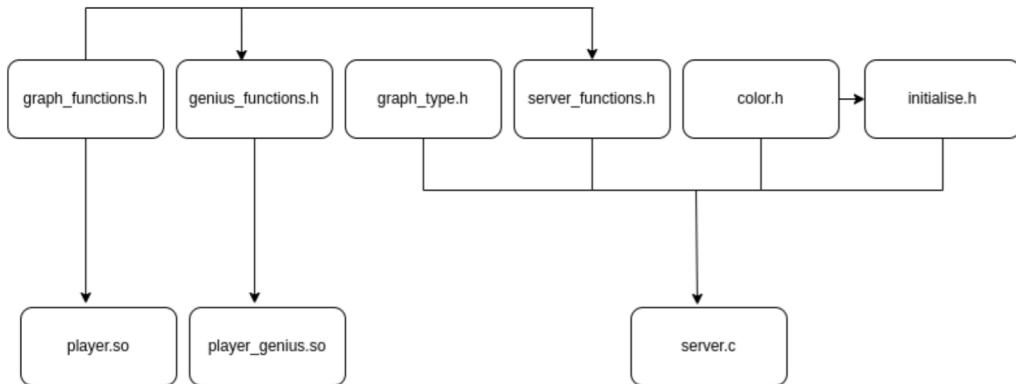


FIGURE 1 – graphe de dépendances

Le diagramme ci-dessus fait état des dépendances des fichiers de notre code. Nous avons choisi de faire des fichiers séparés pour chaque module

- Le fichier `server.c` contient la boucle principale du projet, qui organise la partie entre les joueurs.
- Le fichier `server_functions.h` où on a codé des fonctions utiles pour le fichier `server.c`.
- `graph_type.h` qui contient les fonctions qui génèrent les graphes.
- `initialise.h` qui génère le tableau de couleurs utilisé dans le graphe
- `graph_functions.h` contient les fonctions qui colorie les graphes.
- `genius_functions.h` contient les fonctions utiles pour la fonction min-max

3.2 Architecture des fichiers et détails techniques

Les fichiers dans le dépôt sont organisés de la manière suivante :

```

/
    -- la racine du répertoire du projet
Makefile
    -- Makefile global
README.md
    -- fichier qui contient les instructions d'exécution
/src
    -- fichiers sources (.c/h)
/src/test
    -- les différents tests
/install
    -- Répertoire Install
/rapport
    -- Rapport du projet (.tex/pdf)

```

Le fichier `Makefile` fournit :

- **une règle build** qui compilera l'ensemble du code et créera un exécutable `server`.
- **une règle test** qui compilera les tests sans les exécuter et créera un exécutable `alltests`.

- une règle `install` qui réalisera les règles build et test et copiera les exécutables (server, alltests, et un certain nombre de fichiers .so) à l'intérieur du répertoire install.
- une règle `clean` qui supprimera les fichiers compilés et installés.

Le `Makefile` reçoit une variable `GSL_PATH` qui lui donne le répertoire où est située la bibliothèque `libgsl.so`. Lors de l'exécution, il faut aussi indiquer à l'aide de la variable `LD_LIBRARY_PATH` le répertoire où sont situés les exécutables, dans notre cas `pwd/install`.

4 Implémentation du serveur de jeu

4.1 Initialisation du plateau du jeu

Comme mentionné dans la sous-partie 2.1, un plateau de jeu nécessite l'implémentation à la fois du graphe et des couleurs.

L'implémentation d'un graphe est réalisée dans le fichier `graph.h` sous forme d'une matrice d'adjacence de type `gsl_spmatrix_uint`. Plusieurs types de graphes ont été implémentés, notamment les graphes donut et haché en faisant des trous à l'intérieur d'un graphe carré déjà initialisé.

Néanmoins, la manière dont GSL crée la matrice d'adjacence nous oblige à parcourir tous les sommets pour déterminer les voisins d'un sommet donné. Heureusement, GSL nous permet de convertir cette matrice en une autre qui donne seulement le numéro des voisins pour chaque sommet.

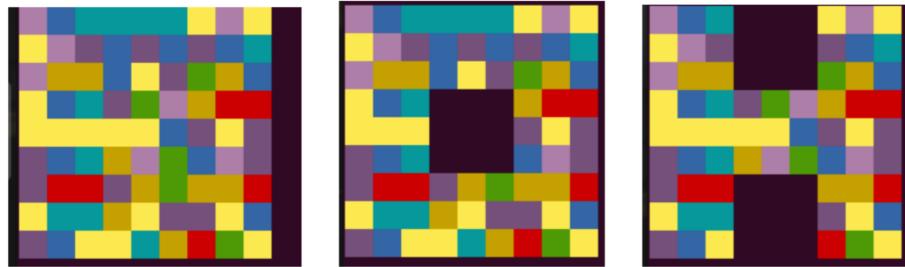


FIGURE 2 – Les différents types des graphes

Complexité

Lors de la création d'un graphe carré, on parcourt tous les sommets et pour chaque sommet, on parcourt à nouveau tous les sommets pour déterminer ses voisins. Donc la complexité est $O(n^2)$ avec n le nombre de sommets dans le graphe.

Le plateau de jeu est initialisé dans une structure `graph_t` décrite dans le fichier `graph.h`. Avant d'initialiser le plateau, il faut récupérer les options, ceci est réalisé grâce à la fonction `parse_opts` qui modifie les valeurs de `board_size`, type de graphe, nombre de couleurs utilisé, nombre de couleurs interdites et l'algorithme de coloration.

En outre, L'implémentation des couleurs est réalisée sous forme d'un tableau `colors`. Deux colorations ont été effectuées : une coloration aléatoire et une coloration mettant en jeu des blocs de sommets colorés.

4.2 Mise à jour du plateau du jeu

Après chaque tour de joueur, il était nécessaire de mettre à jour le plateau du serveur, même les plateaux des joueurs, pour cela on a implémenté une fonction `color_graph` dans le fichier `graph_functions.c` qui sert à attribuer une couleur à chaque case appartenant à un joueur, et pour faire cela on a décidé d'utiliser l'algorithme **Breadth-First Search** ou Algorithme de parcours en largeur. BFS commence à partir d'un noeud et examine tous ses voisins dans un premier temps, puis il examine tous les voisins des voisins, et ainsi de suite...

Puisque chaque sommet est visité au plus une seule fois alors la complexité en temps dans le pire cas l'algorithme est en $O(n)$

Voici les étapes de l'algorithme :

1-Mettre le noeud source dans la file;

2-Retirer le noeud du début de la file pour le traiter et le marquer comme déjà visité;

3-Mettre tous ses voisins non explorés dans la file (à la fin);

4-Si la file n'est pas vide reprendre à l'étape 2.

l'image ci-dessous décrit le fonctionnement de BFS :

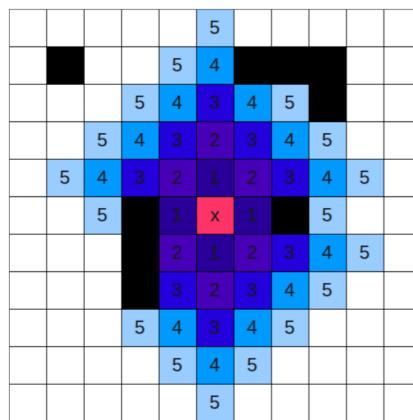


FIGURE 3 – Fonctionnement de l'algorithme BFS sur un graphe

4.3 Boucle de jeu

Tout d'abord, pour commencer la boucle de jeu, il faut charger les joueurs dynamiquement à l'aide de la fonction `init_player_functions` dans `server_functions.c` et puis les initialiser. Ensuite, on initialise 3 graphes et 3 copies de tableaux de couleurs, le premier pair sera utilisé par le serveur les deux autres seront envoyés aux joueurs, afin que ceux-ci ne puissent pas modifier les graphes appartenant au serveur ou à l'autre joueur.

Au sein de cette boucle de jeu, si deux moves successives des deux joueurs ne leur permettent pas de changer de couleurs, alors on récupère le joueur gagnant et on sort de la boucle de jeu. Sinon la fonction `color_graph` va mettre à jour le tableau de couleurs de serveur, puis envoyer ce move à l'autre joueur pour mettre à jour son propre tableau.

Ensuite dans la boucle, il y a une variable `last_move` qui indique si le dernier move était un `NO_COLOR`, c'est-à-dire le joueur passe son tour. Enfin, si les deux joueurs passent leurs tours, on annonce le gagnant par la fonction `is_winning` qui calcule le pourcentage de cases recouvert par chaque joueur et on sort de la boucle du jeu.

Pour finir, on passe au joueur suivant si la partie n'a pas encore terminé en utilisant

`compute_next_player`. On finalise enfin les joueurs et fermons le lien dynamique à l'aide de `dlclose`.

La boucle du jeu correspond au pseudo-algorithme suivant :

Algorithme 1 : Boucle de jeu

```

while i<MAX_NUMBER_OF_MOVES do
    playing = compute_next_player(playing)
    move = player_func[playing].play(move)
    if move is not correct then
        the other player won
        break
    end if
    if move is a no_color and last_move is no_color then
        announce the player winner
        break
    end if
    update_colors
    i+=1
end while

```

4.4 Vérification de la correction des coups

5 Tests du projet

5.1 Les tests

Afin de vérifier le fonctionnement de nos fonctions, on a mis en place des tests des différentes fonctions en utilisant la bibliothèque `assert`. Par la suite, on a analysé la couverture de nos tests avec "gcov".

Plusieurs tests ont mené :

test de graphe

Pour chaque type de graphe, on a créé un graphe de largeur 3 pour pouvoir tester les liaisons pertinentes entre les sommets avant et après l'application de la fonction `make_hole` qui crée des trous à l'intérieur des graphes.

test de player_genius

Suite à la difficulté de tester le fonctionnement de la fonction min-max, on a décidé de se limiter aux tests de l'arbre du jeu.

Pour cela, on a testé la profondeur de l'arbre de jeu pour un arbre à un noeud (racine), un arbre à une profondeur égale à 3 et le même arbre après extension.

De même, on a testé le fonctionnement des fonctions utilisées dans la fonction qui construit l'arbre :

- `color_exist` : cette fonction prend une matrice initialisée à 0 et elle change les cases au bord de celles conquises par le joueur genius à 2. On a donc initialisé un plateau de jeu, on a testé les cases en question.

- `is_connexe` : cette fonction prend une matrice initialisée à 0 et elle change les cases de l'adversaire à 2. De la même manière que la fonction précédente, on initialise un plateau de jeu et on teste les cases en question.

5.2 Les Affichages

Une des préoccupations qu'on a eues au début du projet est de pouvoir visualiser le plateau du jeu. Ainsi, nous avons choisi de faire un affichage dans le terminal afin de suivre l'avancement de la partie pour chaque type de graphe.

6 Problèmes intéressants rencontrés et solutions algorithmiques

6.1 Conversion coo/csr

Lors de la conversion en csr pour les matrices donut et hache, GSL considère que les voisins mis à 0 pour un sommet i comme voisin et donc l'ajoute à la liste des voisins. Pour palier à ce problème, nous avons créé une fonction `gsl_copy` qui, par son nom, crée une nouvelle matrice d'adjacence et copie l'ancienne matrice (après la création du trou) dans la nouvelle. Ainsi, seuls les sommets ayant comme valeur 1 sont considérés comme voisins dans la matrice d'adjacence après conversion.

6.2 La génération des couleurs du plateau de jeu

Pendant le déroulement du projet, nous avons rencontré des problèmes dans l'exécution de certain plateau de jeu. Pour pouvoir détecter et résoudre ce problème, nous avions besoin de régénérer le même plateau, d'où l'idée d'utiliser un `seed` pour initialiser le générateur aléatoire des couleurs des plateaux et donc pouvoir régénérer les plateaux du jeu.

6.3 Implémentation de min-max

Lors de l'implémentation de la fonction min-max, on voulait que min-max renvoie le noeud correspondant au noeud à appliquer. Néanmoins, en testant la fonction min-max, on s'est rendu compte que min-max ne fonctionnait pas correctement. On a donc décidé de changer l'implémentation de la fonction min-max, mais aussi de la structure `node_t` en ajoutant un entier `s` dans la structure `content`. cet entier servira à remonter l'heuristique du noeud correspondant à l'exécution de la fonction min-max.

7 Joueurs et stratégies

Dans ce projet nous avons mis en place 3 joueurs, ceux-ci ont différentes stratégies :

- Stratégie de jeu greedy : Implémentée dans le player `:player.c`
- Stratégie de jeu de paresseuse : Implémentée dans le player `:player_Three.c`
- Stratégie de jeu avancée : Implémentée dans le player `:player_genius.c`

7.1 Description des joueurs

Un joueur est décrit par une structure contenant toutes les informations propres à ce joueur :

```
struct player{
    char *name;
    size_t id;
    struct graph_t *graph ;
```

```

enum color_t *colors;
struct color_set_t forbidden;
};

```

Les joueurs contiennent obligatoirement 3 fonctions `initialize`, `play` et `finalize` ce sont ces fonctions qui sont appelées ensuite dans le serveur et permettra l'interaction entre le serveur et le joueur. Dans `initialize` le joueur devra initialiser la structure `player`.

La fonction `play` doit recevoir un coup puis adapter son graphe au coup reçu et enfin renvoyé un coup adapté à sa stratégie.

Tous nos joueurs mettent ici également à jour le graphe qui leur est propre. En effet, les joueurs doivent avoir un système de mise à jour de leurs graphes, car la seule interaction qu'ils ont avec le serveur après l'initialisation est par le `previous move` envoyé par le serveur.

La fonction `finalize` libère les données utilisées par le joueur, auparavant dans notre cas le graphe et le tableau de couleurs.

7.2 Algorithme de jeu greedy

Un algorithme greedy ou de glouton est une approche pour résoudre un problème en sélectionnant la meilleure option disponible à un moment donné. Il ne se soucie pas de savoir si le meilleur résultat actuel apportera le résultat optimal global. En effet, notre joueur greedy choisit une couleur qui lui permettra de conquérir un maximum d'espace, en effet ce joueur ne respect pas les règles du jeu où il peut choisir une des couleurs interdites. ce joueur choisit la couleur selon la stratégie avec la fonction `find_best_color` dans le fichier `player.c` qui utilise l'algorithme DFS ou l'algorithme de parcours en profondeur.

La figure ci-dessous montre que le joueur greedy `To_The_Moon` qui se trouve sur le dernier sommet a choisi la couleur violette qui lui a assuré un max d'espace.



FIGURE 4 – Démonstration de stratégie de jeu Glouton

7.3 Algorithme de jeu paresseux

L'algorithme consiste à trouver la couleur d'une case la plus proche dont le joueur a le droit de jouer. Le parcours en largeur explore tous les sommets accessibles depuis le sommet source. De plus, lors de ce parcours, les sommets sont explorés par distance croissante au sommet source. Grâce à cette propriété, on a utilisé cet algorithme, le parcours s'arrête lorsque le joueur trouve une couleur différente de ses couleurs interdites.

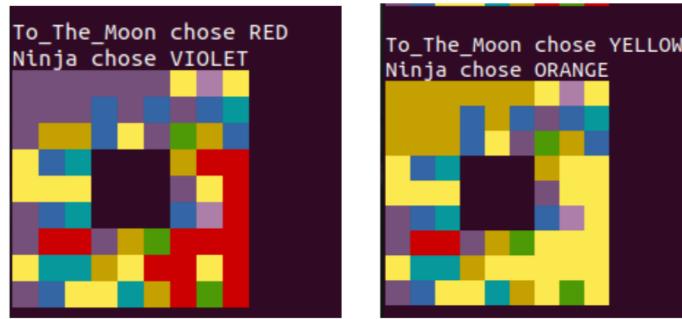


FIGURE 5 – Démonstration de stratégie de jeu Paresseuse

7.4 Algorithmme de jeu avancé

L'algorithme min-max est un algorithme qui s'applique à la théorie des jeux pour les jeux à deux joueurs à somme nulle (les gains réalisés par un joueur sont des pertes pour l'autre joueur) et à information complète (les deux joueurs ont à tout moment toute l'information sur l'état du jeu).

Cet algorithme consiste à minimiser la perte au maximum (c'est-à-dire dans le pire des cas).

7.4.1 Arbre de jeu

l'algorithme min-max de base sur la création d'un arbre de jeu qui trace tous les coups possibles en partant d'une situation courante et pour une profondeur limitée. Cet arbre est construit en alternant les deux joueurs comme l'illustre l'image ci-dessous :

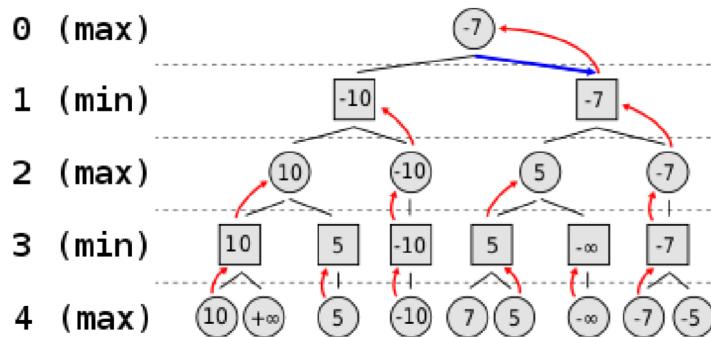


FIGURE 6 – arbre min-max de profondeur 4

Pour chacune de ces situations, si elle est choisie par le joueur, elle devient la prochaine situation courante à partir de laquelle l'autre joueur devra jouer et ainsi de suite jusqu'à la profondeur choisie ou la fin des choix possibles.

Comme évoqué précédemment, cet algorithme s'applique aux jeux à somme nulle ce qui nous emmène au choix de l'heuristique. Ainsi, on choisit le pourcentage des cases conquises par le joueur parmi les cases conquises par les deux joueurs comme heuristique.

Deux structures ont été implémentées afin de tracer l'arbre de jeu :

```
struct node_t {
```

```

struct node_t* parent;
struct node_t* children[8];
struct content* c;
int is_initialised;
};

struct content{
    enum color_t *colors;
    int p;
    int s;
};

```

une pour représenter un nœud de l'arbre et une autre qui contient le tableau couleur et l'heuristique pour chaque nœud.

7.4.2 Principe

À l'aide d'une recherche récursive, on parcourt l'ensemble des coups possibles. On associe à chacun des joueurs un entier. Le joueur qui utilise l'algorithme min-max se nomme joueur 1 et l'autre joueur 0. Le but du joueur 1 est de maximiser le score, à l'inverse, le joueur négatif doit le minimiser.

Pour l'état courant, on teste si cet état est final, auquel cas, nous renvoyons le nœud correspondant à cet état et nous mettons dans le champ s du context l'heuristique de ce nœud. Si l'état n'est pas final, nous réitérons cette recherche pour tous états fils en distinguant les joueurs et nous retournons le nœud fils dont le champ s est maximal (respectivement minimal) pour le cas du joueur 1 (respectivement joueur 0). Comme le montre la figure 1, pour chaque nœud, nous mettons, dans le champ s du context du noeud en question, le maximum des champs s de tous les nœuds fils dans le cas d'un noeud max (c'est-à-dire le nœud qui correspond au move du joueur en question) et le minimum dans le cas d'un noeud min (c'est-à-dire le nœud qui correspond au move du joueur adversaire).

```

function minimax(player , node , joueur , depth):
    if depth = 0 or node is a terminal node then
        node->c->s=node->c->p
        return the node
    if joueur=1 then
        value <0
        for each child of node do
            value := max(value , heuristique(minimax(player , child , depth-1, 0)))
            node->c->s=value
    else if joueur=0
        value >>0
        for each child of node do
            value := min(value , heuristique(minimax(player , child , depth-1, 1)))
            node->c->s=value
    return value

```

8 Discussion

8.1 Limites et améliorations possibles

algorithme min-max :

Le temps nécessaire à la réalisation de l'algorithme min-max est tel qu'il est impossible, dans la plupart des cas, d'atteindre la fin de partie. En effet, la complexité en temps d'un tel algorithme est exponentielle par rapport à la profondeur nécessaire pour atteindre la fin de la partie.

Pour palier à ce problème, nous avons choisi d'initialiser l'arbre à une profondeur 6 et après avoir avancé dans l'arbre, on libère la partie de l'arbre qui n'est pas choisi et on étend l'arbre d'une profondeur.

En outre, cet algorithme peut être optimisé grâce à l'implémentation de la technique dite de l'élagage alpha-bêta. L'algorithme alpha bêta accélère la routine de recherche min-max en éliminant les cas qui ne seront pas utilisés. Cette méthode utilise le fait que tous les autres niveaux de l'arbre seront maximisés et que tous les autres niveaux seront minimisés.

graphe torique

Faute de temps, on n'a pas pu terminer l'implémentation de la partie utilisant le graphe torique. Néanmoins, on a implémenté le graphe torique sans aboutir à son utilisation.

Ce manque de temps est principalement dû à la difficulté qu'on a eu lors de l'implémentation de la fonction min-max évoquée précédemment.

8.2 Problèmes rencontrés

Difficulté rencontrée lors de la réalisation du projet

Nous avons rencontré des difficultés à nous adapter aux éléments que nous ne maîtrisions pas : écriture d'un Makefile, utilisation des bibliothèques dl ou encore l'utilisation de Valgrind et gdb. Cependant, nous avons appris à les maîtriser au fur et à mesure.

9 Conclusion

Dans le cadre des projets de programmation, nous avons donc réalisé un programme du jeu Flood. Ce projet a permis de mettre en œuvre les connaissances que nous avons acquises dans le cours de programmation impérative et de les améliorer. Nous avons été confrontés à de nombreux problèmes, mais nous avons pu trouver des solutions alternatives pour les résoudre. Le travail en groupe était enrichissant pour la confrontation d'idées et a été une bonne manière de progresser, à la fois sur le code et sur la manière d'appréhender et de résoudre les problèmes.