ANALYSE ET RÉSOLUTION DE CASSE-TÊTES : FLOOD-IT

Réalisé par :
BELLEL Oussama
BOURGIN Jérémy
HERRI Abdallah

Sous la direction de :
MASSEPORT Samuel
VOLTE Gabriel

Pour l'obtention du Master1 AIGLE
Année universitaire 2018/2019

Table des matières

Table des figures

Liste des tableaux

Remerciement

Nous tenons à remercier Samuel Masseport et Gabriel Volte pour nous avoir encadrés et conseillés tout au long du projet. En effet, ils ont su bien nous guider pour nous mener à la réussite ce projet.

Chapitre 1

Introduction

1.1 Généralités

Dans le cadre du projet de TER en première année de Master AIGLE, nous devions réaliser un programme permettant de résoudre un casse-tête. Les encadrants de ce projet nous ont laissé le libre choix du casse-tête.

Après avoir essayé différents jeux, nous avons choisi le jeu Flood-It. Le choix de ce jeu s'explique par le fait qu'il permet d'avoir un nombre important d'approches différentes pour le résoudre. Ainsi avec plusieurs méthodes de résolution, le but serait de pouvoir en réutiliser plusieurs sur d'autres casse-têtes.

Pour cela, nous avons joué plusieurs parties et essayé différentes techniques pour finir le jeu le plus rapidement possible. Ainsi, avant même de commencer à implémenter le jeu, nous avions quelques idées d'algorithmes que nous pourrions utiliser pour ce jeu.

1.2 Flood-It

Flood-it est représenté par une grille de M*M cases et N couleurs où chaque case a une couleur générée aléatoirement au début de la partie. C'est un jeu combinatoire : il existe plusieurs combinaisons possibles pour résoudre ce jeu.

Avant d'entrer dans les détails du jeu, nous allons d'abord introduire le terme "zone". Une zone est un ensemble où toutes les cases adjacentes (horizontalement et verticalement) ont la même couleur. La taille de chaque zone de la grille a une borne minimale de 1 (une zone qui contient une seule case) et une borne maximale de M*M.

À chaque tour, le joueur va effectuer une opération de coloriage. Cette opération consiste à changer la couleur de toute la zone où se situe la cellule en haut à gauche (que nous appellerons la "zone initiale"). Cela a pour effet de relier la zone nouvellement colorée à toutes les zones voisines de cette couleur.

L'objectif général est de colorer totalement la grille en effectuant le moins de coloration possible. Notons qu'il n'est pas possible de perdre à ce jeu puisque qu'il est toujours possible de colorier la grille.

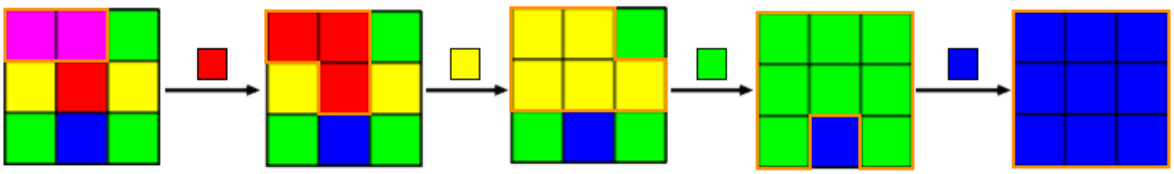


Figure 1.1 – Coloration d'une grille de 3x3

La figure 1.1, représente une séquence optimale de mouvements pour colorer une grille de 3×3.

1.3 Objectifs du projet

La tâche la plus compliquée n'est pas seulement d'implémenter des algorithmes permettant de résoudre Flood-It, mais d'implémenter ces algorithmes de façon à ce que nous puissions les réutiliser sur d'autres casse-têtes. Dès lors, plusieurs problématiques se posent :

- Comment appliquer des techniques de résolution dans le but de pouvoir les ré-appliquer sur d'autres casse-têtes ?
- Comment modéliser notre programme pour faciliter l'implémentation d'un nouveau casse-tête ?
- Sachant que ces casse-têtes sont généralement des problèmes NP-difficiles, chercher à avoir une solution la plus proche possible de la solution optimale, peut s'avérer coûteux en temps et en espace, comment optimiser au maximum nos algorithmes ?
- Comment comparer l'efficacité des différents algorithmes ?

Maintenant que nous avons bien défini les objectifs de notre projet, nous pouvons établir la liste des différentes tâches que nous devons réaliser :

- modéliser et développer le jeu,
- implémenter une interface graphique du jeu,
- développer une approche exacte de résolution,
- développer différentes méthodes d'approche pour la résolution,
- faire un benchmark complet afin de comparer nos différents algorithmes,
- analyser et interpréter les résultats issus du benchmark,
- bonus : implémenter un autre casse-tête sur lequel nous allons réutiliser des méthodes de résolution

À noter que parmi les différentes tâches que nous devons réaliser, il est impératif que le jeu, l'interface graphique et les différents algorithmes fonctionnent indépendamment. Cela est important pour la réutilisation. De plus, si les algorithmes et le jeu fonctionnent de façon dépendante, alors il sera impossible de pouvoir développer un autre casse-tête en utilisant des méthodes de résolutions communes.

Chapitre 2

État de l'art

2.1 Flood-It

Ueverton dos Santos Souza et al.[8] ont analysé le comportement du jeux Flood-It sur d'autres classes de graphes, tels que les d-boards et les grilles circulaires. Ils ont décrit des algorithmes de temps polynomiaux pour jouer à Flood-it sur les 2*n grilles circulaires et certains types de d-boards (grilles à monochromes colonne).

Kitty Meeks et al.[9] ont étudié le jeu combinatoire Flood-It, généralisé aux graphes. Leur objectif est de calculer le nombre de coups nécessaires pour colorer une grille. Ils ont prouvé que Flood-It est résolu en temps polynomial sur des grilles 1*n.

2.2 Métaheuristique

2.2.1 Définition

Une métaheuristique est définie sur wikipedia comme suit : "Une métaheuristique est un algorithme d'optimisation visant à résoudre des problèmes d'optimisation difficile (souvent issus des domaines de la recherche opérationnelle, de l'ingénierie ou de l'intelligence artificielle) pour lesquels on ne connaît pas de méthode classique plus efficace" [2].

Les métaheuristiques comme les heuristiques, essayent de trouver une solution qui s'approche le plus possible de la solution optimale. Les métaheuristiques utilisent pour cela des fonctions avec un haut niveau d'abstraction (donc ces fonctions devront être implémentées par l'utilisateur). Cela leur permet d'être adaptées à tous les problèmes d'optimisation.

2.2.2 Principales caractéristiques

— Les métaheuristiques sont des heuristiques applicables à tous les problèmes d'optimisation.

— Les métaheuristiques sont des stratégies qui permettent de tendre vers la solution optimale.

— Le but visé par les métaheuristiques est d'explorer l'espace de recherche efficacement afin de déterminer des solutions approchées.

— Les techniques qui constituent des algorithmes de type métaheuristique vont de la simple procédure de recherche locale à des processus d'apprentissage complexes.

— Les métaheuristiques ne donnent aucune garantie d'optimalité.

— Les métaheuristiques peuvent contenir des mécanismes qui permettent d'éviter d'être bloqué dans des régions de l'espace de recherche. Les concepts de base des métaheuristiques peuvent être décrits de manière abstraite, sans faire appel à un problème spécifique.

— Les métaheuristiques peuvent faire appel à des heuristiques qui tiennent compte de la spécificité du problème traité, mais elles sont contrôlées par une stratégie de niveau supérieur.

— Les métaheuristiques peuvent faire usage de l'expérience accumulée durant la recherche de l'optimum, pour mieux guider la suite du processus de recherche [3].

2.2.3 Classification

— Méthodes de trajectoire : manipulent un seul point à la fois et tentent toujours de l'améliorer. Par exemple : la recherche tabou.

— Méthodes qui travaillent avec une population de points : en tout temps nous disposons d'une "base" de plusieurs points, appelée population. L'exemple le plus connu est l'algorithme génétique. La figure 2.1 représente le principe général des métaheuristiques (a) à population et (b) à parcours.

— Méthodes avec ou sans mémoire : utilisent l'historique de leurs recherche pour guider l'optimisation aux itérations suivantes.

— Selon leur manière d'utiliser la fonction objectif : certaines métaheuristiques dites statiques travaillent directement sur f alors que d'autres, dites dynamiques, font usage d'une fonction g obtenue à partir de f en ajoutant quelques composantes qui permettent de modifier la topologie de l'espace des points.

— Nombre de structures de voisinage : la plupart des métaheuristiques utilisées dans le cadre des problèmes d'optimisation combinatoire utilisent une seule structure de voisinage. Cependant, des méthodes comme la recherche à voisinage variable permettent de changer de structure en cours de recherche [3].

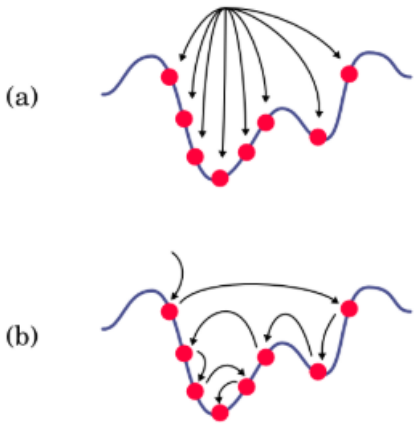


Figure 2.1 – Principe général des métaheuristiques (a) à population, et (b) à parcours.

2.3 Algorithmes gloutons

2.3.1 Définition

D'après la définition de Hao et al.[7] un algorithme glouton (greedy algorithm en anglais, parfois appelé aussi algorithme gourmand). C'est "un algorithme qui suit le principe de faire, étape par étape, un choix optimum local, dans l'espoir d'obtenir un résultat optimum global".

Dans le cadre de notre projet, l'heuristique consiste, étape par étape de jouer un coup considéré comme étant le meilleur. Le but étant d'essayer de se rapprocher du score optimal.

Voici un exemple générique de l'algorithme glouton (donc, selon le problème, il peut y avoir des variantes) :

2.3.2 Algorithme générique d'une procédure gloutonne

Algorithm 1 Glouton

```
S ← ∅
C ← ensemble des candidats à la solution
while S n'est pas une solution et C <> ∅ do
  x ← choisir un élément de C le plus prometteur
  C ← C - x
  if réalisable (solution, x) then
    solution ← solution ∪ x
  end if
end while
if S est une solution then
  return S
else
  return ∅
end if
```

2.3.3 Exemple

Considérons le système de rendu de monnaie, comportant des pièces et des billets. L'application de l'algorithme glouton permet de rendre la monnaie avec le minimum des pièces qui sont à sa disposition. Par exemple, la meilleure façon de rendre 7€ :

Considérons que le système possède des billets de 10 et 5€, des pièces de 2 et 1 €, des pièces de 50, 20 et 10 centimes.

Le système commence par la valeur la plus grande (billet de 10€). C'est impossible, car la somme qu'il souhaite rendre est supérieur à 7€. Donc il passe à la valeur qui suit et doit donc rendre 5€. Il reste donc 2€ à rendre. Ensuite, il vérifie à nouveau s'il peut rendre 5€. C'est impossible puisqu'il reste 2€. Le système va donc essayer avec la valeur qui suit. Il peut donc rendre 2€. Il ne reste plus rien à rendre, le système a terminé.

Au final, le système retourne un billet de 5€ et une pièce de 2€. le tableau 2.1 ci-dessous montre les traces d'exécution de l'algorithme

/	C	X	S
0	Billets : 10 et 5 €, pièces : 2 et 1 €, et 50,20 et 10 centimes	not defined	∅
1	Billet : 5 €, pièces : 2 et 1 €, et 50,20 et 10 centimes	Billet 10 €	∅
2	Billet 5 et pièces : 2 et 1 €, et 50,20 et 10 centimes	Billet 5 €	[Billet 5 €]
3	pièces : 2 et 1 €, et 50,20 et 10 centimes	Billet 5 €	[Billet 5 €]
4	pièces : 1 €, et 50,20 et 10 centimes	pièces 2 €	[Billet 5 €, pièce 2 €]

Table 2.1 – Trace d'exécution de l'algorithme glouton

2.4 Recherche tabou

2.4.1 Définition

D'après wikipedia : "la recherche tabou est une métaheuristique d'optimisation présentée par Fred W. Glover (en) en 1986. On trouve souvent l'appellation recherche avec tabous en français. Cette méthode est une métaheuristique itérative qualifiée de recherche locale au sens large" [5].

L'algorithme consiste à parcourir ses voisins de manière complète, de rechercher un minimum optimal, et dès qu'une meilleure solution est trouvée, elle remplace la solution courante. Cependant, lorsqu'il n'y a plus de meilleure solution dans le voisinage, le meilleur voisin remplace la solution courante même si celui-ci est moins bon puis la recherche recommence : nous dégradons la solution courante. Pour cela, il existe une liste tabou permettant de ne pas boucler. Nous ajoutons successivement les actions effectuées dans la liste et nous nous interdisons de refaire une action se trouvant dans celle-ci, car nous disons que l'action devient "tabou". Notons que cela fonctionne aussi pour les problèmes de maximisation, à la différence que l'on cherchera un maximum local.

2.4.2 Algorithme de la recherche tabou

Algorithm 2 Recherche Tabou

Ensure : Une solution meilleure ou égale à Sol
  Initialisation des mémoires à court, moyen et long terme
  repeat
    $Sol' \leftarrow$ meilleur voisin de Sol
    MAJ de la liste tabou, critère d'aspiration et des mémoires à moyen et long terme
    if critère d'intensification est vérifié **then**
      Intensification
    end if
    if critère de diversification est vérifié **then**
      Diversification
    end if
  until le critère d'arrêt soit vérifié
  return la meilleur solution trouvée

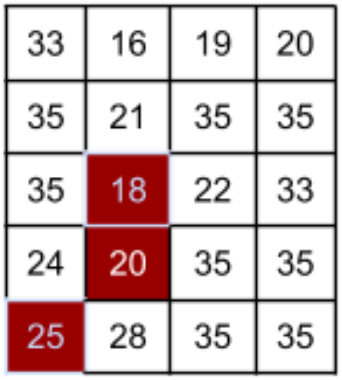
2.4.3 Exemple

Dans cet exemple, le but du jeu est de rechercher la valeur minimale dans un tableau en essayant d'effectuer le moins de déplacement possible. Pour cela, nous devons commencer par la case qui est en bas à gauche. Le déplacement se fait de case en case où le voisinage se situe sur l'axe horizontal, vertical et diagonal.

La liste tabou est utilisée pour stocker les cases par lesquelles nous passons. Nous avons choisi un exemple où il n'y a pas de risque de boucler. Par conséquent, la liste tabou correspond simplement au parcours effectué.

L'exemple ci-dessous illustre le fonctionnement de l'algorithme sur une grille de 4x5. Ici, nous accepterons de dégrader la solution courante avec un seuil maximum de 5.

l'algorithme commence à chercher la valeur minimale. Pour cela, nous commençons la recherche sur la cellule en bas à gauche. Ici nous initialisons nos 2 variables :
optimal = 25
parcours = []



l'algorithme parcours [25→20→**18**]. Sachant que 18 < 25, alors optimal = 18
(l'algorithme glouton se serait s'arrêter ici)

l'algorithme ne trouve pas une valeur inférieur à 18, donc il est obligé de dégrader sa solution et de prendre le parcours [25→20→18→**21**→16]. Sachant que 16 < 18, alors optimal = 16

l'algorithme dégrade la solution une autre fois et prend le parcours [25→20→18→21→16→**19**→**20**], Finalement, l'algorithme s'arrête ici puisque les voisins de la cellule courante dégraderont trop la solution. Ici le résultat n'est pas meilleur que l'optimal. Donc le résultat final est :
optimal = 16
parcours = [25→20→18→21→16]

Figure 2.2 – Illustration de l'algorithme tabou

Chapitre 3

Conception

3.1 Modélisation

Une fois que nous avions implémenté une version jouable de Flood-It, il nous fallait un modèle réutilisable permettant de pouvoir analyser le plus simplement possible le jeu. Pour cela, nous avons commencé à modéliser le jeu de façon à ce qu'il soit fidèle à la réalité, et que l'on puisse facilement analyser le voisinage de la zone initiale. Voici le diagramme de classe que nous avons réalisé pour cela :

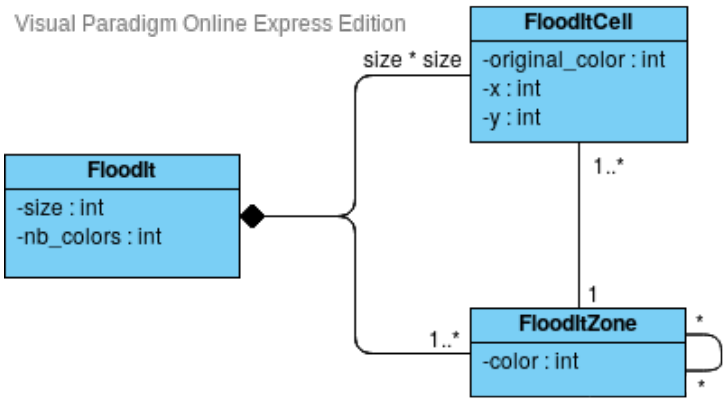


Figure 3.1 – Diagramme UML de FloodIt

Ici, nous avons le jeu (FloodIt) qui est représenté par la taille de la grille et le nombre de couleurs. Il est composé de :

- l'ensemble des cellules du jeu (donc une matrice),
- l'ensemble des zones du jeu.

Les cellules sont représentées par leur position x y dans la matrice, et leur couleur originale. La couleur originale nous permet de pouvoir :

- créer les zones à l'instanciation du jeu (puisque nous devons d'abord créer les cellules et leur attribuer une couleur aléatoire),

— recommencer une partie.

Enfin, les cellules ont une référence vers la zone dans laquelle elles se situent. Les zones sont caractérisées par la couleur courante des cases la composant. Les zones possèdent donc l'ensemble des cellules qui les compose et les zones voisines qui leur sont adjacentes.

Ainsi, nous allons analyser l'état du jeu en fonction des zones adjacentes à la zone initiale. En effet, s'intéresser au voisinage des cases est plus compliqué :

— cela nécessite de faire de multiples appels de fonctions récursifs,

— de déterminer la zone dans laquelle la cellule se situe,

— etc...

Alors qu'analyser les zones et leur voisinage est bien plus simple. De plus, cela donne beaucoup plus d'informations.

Maintenant que nous avons un modèle permettant d'analyser l'état du jeu, nous devons dresser un modèle permettant aux heuristiques d'appliquer des méthodes de résolution sur celui-ci. Il faut donc que le modèle permette :

— le fonctionnement des heuristiques sur d'autres jeux,

— l'ajout d'autres heuristiques sans impacter les autres.

Pour cela, nous avons commencé à dresser un schéma avec les premières méthodes de résolution que nous voulions implémenter :

— la recherche naïve,

— la recherche exhaustive,

— la recherche gloutonne.

Par conséquent, nous avons besoin de savoir comment nous pourrions obtenir des informations sur le jeu au sein de nos méthodes de résolution. À cet effet, les méthodes suivantes seront nécessaires :

— récupérer la liste des coups possibles,

— jouer un coup,

— savoir si la partie est terminée,

— évaluer un coup (nous avons trouvé plusieurs manières pour évaluer si un coup sera meilleur qu'un autre ou non).

Maintenant que nous savons comment résoudre un jeu dans différentes méthodes de résolution, nous avons commencé par effectuer des recherches nous permettant de trouver un modèle réutilisable. Nous avons donc dressé un premier diagramme UML :

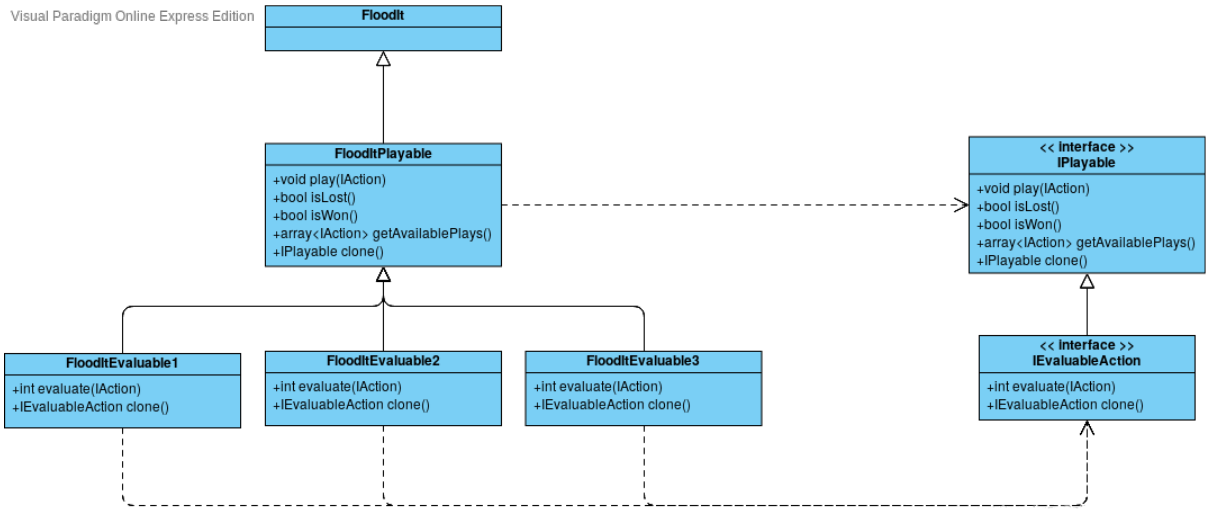


Figure 3.2 – Diagramme UML - version 1

Dans ce modèle, nous remarquons que :

- le schéma est vertical (les "Evaluable" héritent des "Playable" qui héritent du jeu),
- le schéma est symétrique (les classes implémentant une interface A, héritent des classes implémentant une interface B lorsque l'interface A hérite de l'interface B),
- le schéma se découpe en plusieurs couches (les interfaces et les classes héritent de façon successive : couche 0 = FloodIt, couche 1 = Playable, couche 2 = Evaluable)

Avec ce schéma, nous pouvons appliquer les heuristiques sur d'autres jeux. Par contre, lorsque nous ajoutons une nouvelle heuristique, cela risque d'avoir un impact sur les autres. Si l'ajout d'une heuristique nécessite l'introduction de nouvelles méthodes (et donc une nouvelle interface), cela peut apporter des complications. En effet, lorsqu'il y a plusieurs versions de l'implémentation d'une interface pour un jeu (dans notre cas nous avons plusieurs versions "Evaluable"), l'ajout d'une interface dans une nouvelle couche induira la création d'un nombre important de classes identiques. Effectivement, elles devront hériter de chacune des classes de la couche du dessus. Cela fera donc évoluer le nombre de classes de façon drastique. Ainsi, le risque d'impacter le fonctionnement des heuristiques survient lorsque l'on doit ajouter une interface entre 2 couches. En effet, des modifications à apporter sur la couche du dessous sont nécessaires. Cela risque donc d'apporter l'introduction de bugs lors de la résolution du jeu. Comme le dit Edsger Dijkstra : "Si debugger, c'est supprimer des bugs, alors programmer ne peut être que les ajouter" [14]

Nous avons donc essayé de réaliser un modèle permettant de rendre les heuristiques indépendantes. Voici la deuxième version du diagramme de classe que nous avons effectuée :

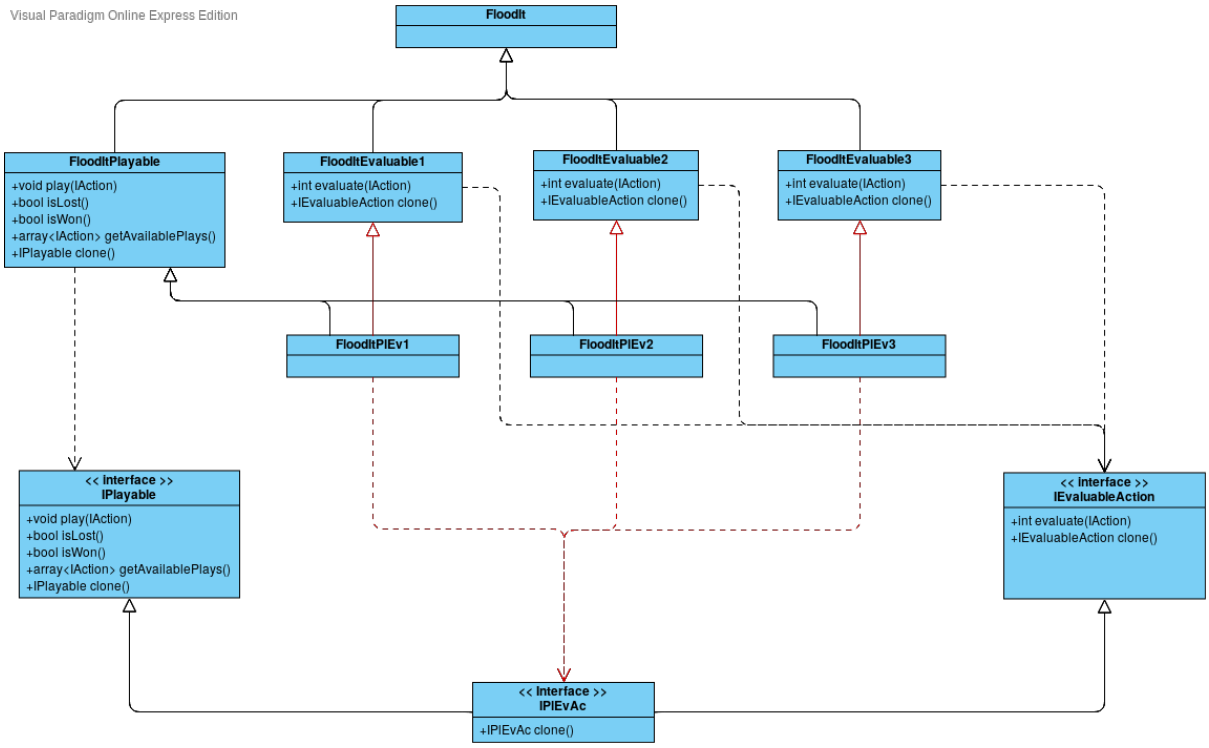


Figure 3.3 – Diagramme UML - version 2

Dans ce modèle, nous remarquons que :

- le schéma est symétrique,
- il y a exactement 3 couches (couche 0 = FloodIt, couche 1 = Playable, Evaluable, couche 2 = PlayableAndEvaluable),
- le schéma est horizontal (on ajoute simplement les interfaces et l'implémentation de celle-ci dans la deuxième couche),
- les classes de la troisième couche permettent uniquement de satisfaire des interfaces en héritant de différentes classes (FloodItPlEv1 hérite de FloodItplayable et FloodItEvaluable1 afin d'avoir un modèle Playable et Evaluable)
- il y a beaucoup de dépendances

Avec ce schéma, nous pouvons donc appliquer des heuristiques sur d'autres jeux et ajouter de nouvelles heuristiques sans impacter les autres (elles sont donc indépendantes). Cependant, il y a 2 problèmes majeurs :

- il y a un nombre important de dépendances, ce qui rend la maintenance compliquée et risque d'impacter les performances,
- il y a beaucoup de classes créées uniquement dans un but déclaratif, ce qui rend le projet plus complexe qu'il n'y parait (de plus, il est pénible d'écrire un nombre important de classes où leur but est uniquement de faire l'héritage)

Bien que ce schéma permette la réutilisation des heuristiques, il comporte toutefois des inconvénients significatifs. Par conséquent, nous avions décidé de nous documenter sur les différents patrons de conception existants et sur les possibilités que peut offrir le paradigme orienté objet. Nous avons donc découvert qu'il était possible d'hériter d'un type paramétré. Ensuite, nous avons voulu implémenter un schéma de conception en utilisant cette possibilité. Cependant, nous nous sommes retrouvés confrontés à divers problèmes techniques. Par conséquent, nous avons effectué des recherches sur l'héritage paramétré. Nous avons découvert qu'il existait la patron de conception "mixin". Il permet de palier à diverses problématiques :

— le problème de l'héritage en diamant [15],

— permet de faire de l'héritage multiple [16],

— permet de factoriser des fonctionnalités issues de différentes classes [17]

Pour cela, nous avons dressé le schéma suivant :

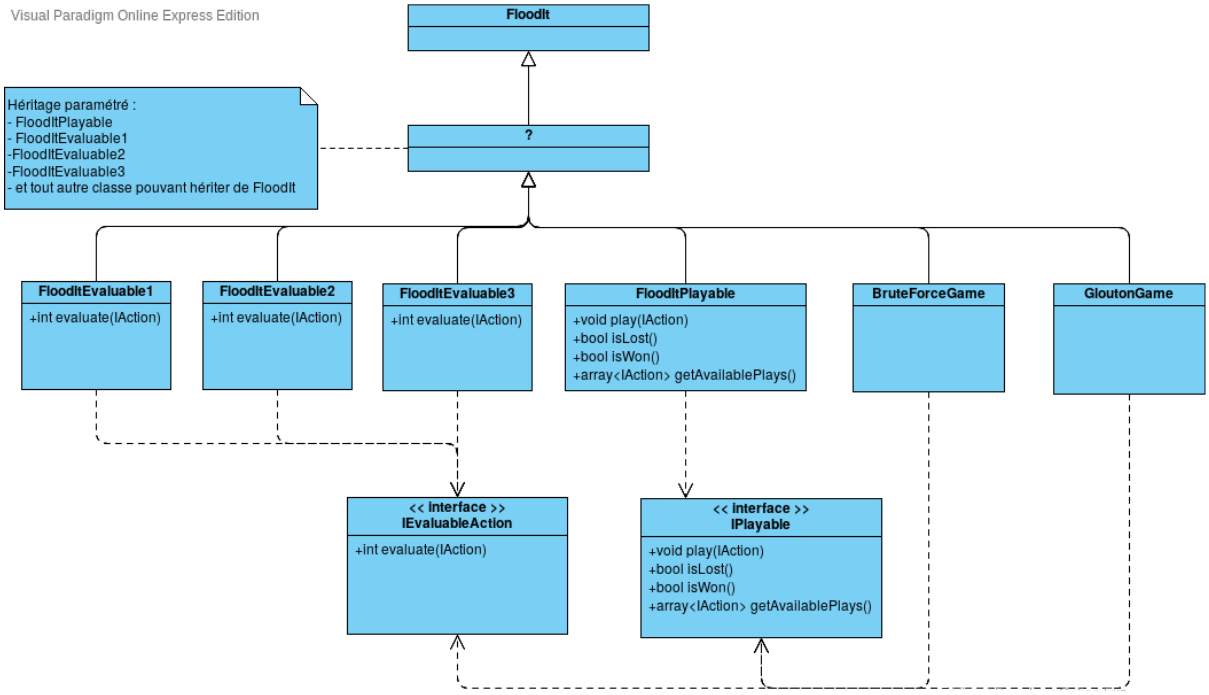


Figure 3.4 – Diagramme UML - version 3

Dans ce modèle, nous remarquons que :

— il y a 3 couches (couche 0 = FloodIt, couche 1 = Playable, Evaluable, couche 2 = BruteForceGame, GloutonGame)

— la troisième couche sont des classes qui permettent de spécifier les méthodes à implémenter pour chaque heuristique (GloutonGame doit hériter de classes permettant d'être Playable et Evaluable)

— il n'y a plus le problème du clonage (on peut directement utiliser le constructeur par copie des classes de la troisième couche dans les heuristiques) [18]

Avec ce schéma, nous définissons une chaîne d'héritage pour une instance donnée (ce qui revient donc à faire de l'héritage multiple partagé). La classe finale doit être une classe de la troisième couche et doit hériter des classes permettant de pouvoir implémenter toutes les méthodes spécifiées dans les différentes interfaces dont elles dépendent. Par exemple, pour créer une instance d'un jeu que l'heuristique gloutonne pourra utiliser, nous écrirons :

```
size_t size = 20;
size_t nb_colors = 5;
GloutonGame<
    FloodItEvaluable1<FloodItPlayable<>>
> glouton_game (size, nb_colors);
```

Dans cet exemple, GloutonGame (donc la classe finale) hérite de FloodItEvaluable1, qui hérite de FloodItPlayable et qui hérite de FloodIt. Par conséquent, cela "génère" un schéma semblable à celui de l'exemple 1, à la différence qu'il est beaucoup plus modulable comme dans l'exemple 2. Cependant, il permet d'avoir une architecture où le nombre de classe n'évolue pas de façon importante (il existe seulement à but classe déclaratif par heuristique ou un nombre limité), et où il y a peu de dépendance (d'ailleurs il n'y a pas d'héritage entre les interfaces). De plus, il est plus simple d'utilisation et permet de résoudre les problèmes de typage statique et dynamique des méthodes (sans risque de problème à l'exécution puisqu'avec la généricité tout est vérifiée à la compilation). En effet, nous pouvons utiliser la généricité pour spécifier quelle heuristique sera utilisée, avec quel jeu et les classes implémenter. Par exemple :

```
// fonction paramétré
// Herustic = classe implémentant une heuristique
// Game = classe de jeu destiné à une heuristique (troisième couche)
// Base = chaîne d'héritage
template<
    template<class> class Herustic,
    template<class> class Game,
    class Base
>
void make_herustic(size_t size, size_t nb_colors)
{
        Game<Base>* game = new Game<Base>(size, nb_colors);
        Herustic<Base>* herustic = new Herustic<Base>(*game);
        ...
}

...

// utilisation de la fonction
make_herustic<
    Glouton,
    GloutonGame,
    FloodItEvaluableActionTry1<FloodItPlayable<>>
>(20, 5);
```

Pour finir, nous avons donc décidé d'utiliser la patron de conception "mixin" puisqu'il nous permet de pouvoir avoir des heuristiques réutilisables sur d'autres jeux et les heuristiques indépendantes entre elles. En effet, il permet plus de modularité, que le projet soit moins

complexe, d'éviter les bugs (puisque tout est vérifié à la compilation), une meilleure factorisation du code et une maintenance plus simple. De plus, les possibilités qu'il entraîne sont intéressantes sachant que son utilisant est très peu répandu.

3.2 Recherche naïve

Pour la recherche naïve, nous avons décidé d'implémenter une méthode qui joue les coups de façon aléatoire. Pour cela, à chaque itération, elle récupère la liste de tous les coups possibles, et en choisi un aléatoirement.

Cela nous permet donc d'avoir un algorithme de comparaison. En effet, si une heuristique obtient des résultats similaires à celui-ci, alors nous pouvons conclure qu'elle est mauvaise.

3.3 Recherche exhaustive

La recherche exhaustive (appelée recherche par force brute) est une méthode algorithmique qui permet d'essayer toutes les solutions possibles pour arriver à la meilleure solution. Cependant, cette technique nécessite des ressources matérielles et prend beaucoup de temps pour effectuer une résolution.

Dans le cadre de notre projet, l'arbre de recherche est défini de la manière suivante : chaque noeud de l'arbre correspond à l'état du jeu courant, le voisinage de chaque noeud correspond à tous les coups possibles pour celui-ci (donc les feuilles de cet arbre sont des jeux résolus). La hauteur minimale de cet arbre est une succession de coup pour lequel on résout le jeu en un minimum de coup. Pour cela, la recherche exhaustive va effectuer un parcours en profondeur dans cet espace de recherche.

Nous avons décidé d'utiliser ce parcours, car il permet d'utiliser le moins de mémoire possible. En effet, nous avons besoin uniquement d'un nombre de copie du jeu (où une copie du jeu correspond à un noeud de l'arbre) égale à la hauteur de celui-ci. En effet, nous pouvons libérer la mémoire au fur et à mesure que l'on remonte dans l'arbre.

Nous allons illustré le fonctionnement de cette méthode avec un exemple. La figure 3.5 ci-dessous représente une grille de 4x4 avec 4 couleurs :

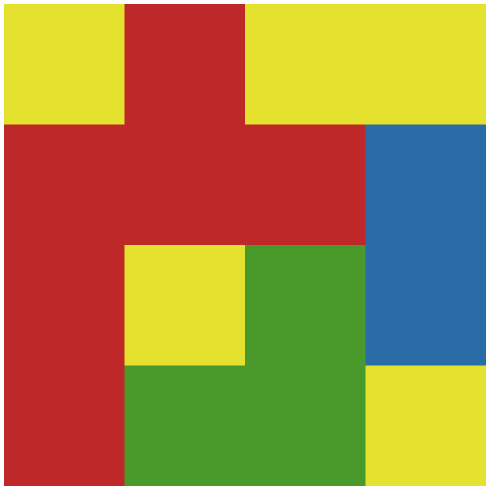


Figure 3.5 – Recherche exhaustive - Grille

La figure 3.6 ci-dessous illustre l'arbre de recherche complet pour notre exemple où un noeud représente l'état du jeu courant, un carré représente un coup joué et les feuilles colorées correspondent au plateau final.

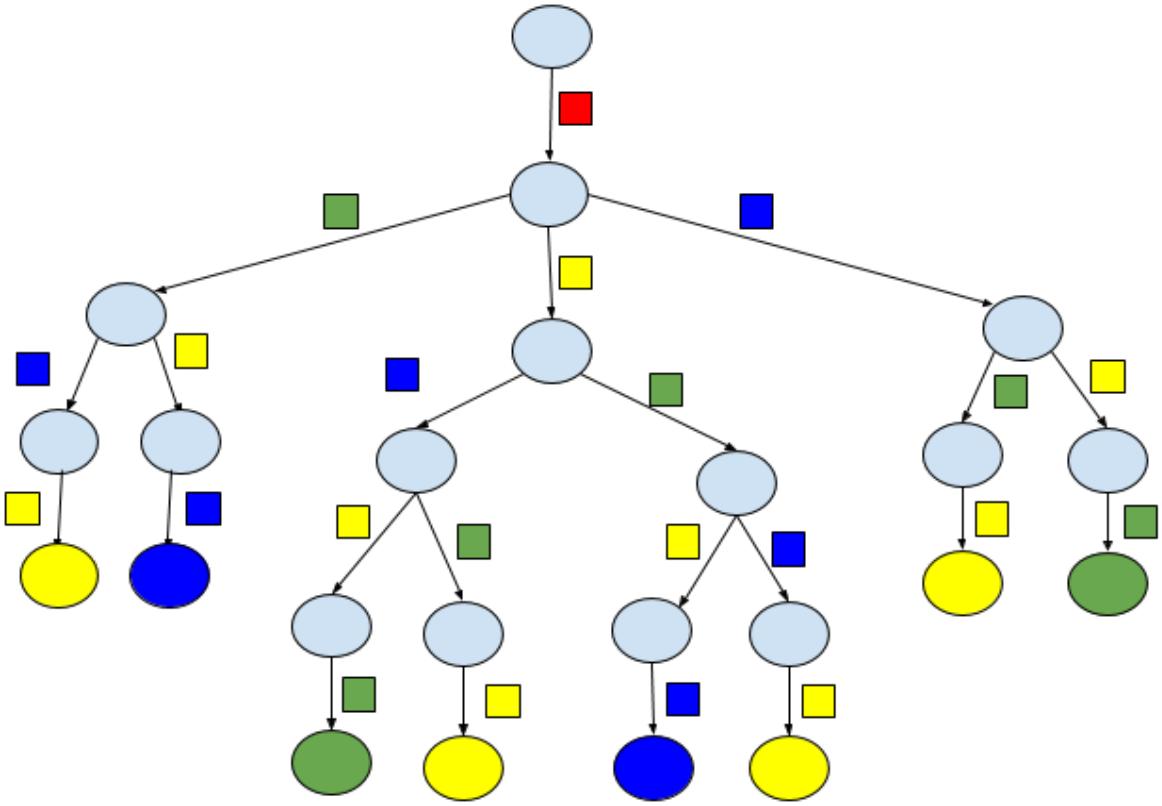


Figure 3.6 – Illustration de l'arbre de recherche

La recherche exhaustive va donc effectuer un parcours en profondeur dans le but de récupérer la succession de coup optimale. Voici le résultat obtenu :
Rouge→Vert→Bleu→Jaune

3.4 Algorithme glouton

Glouton est une façon générale de résoudre les problèmes d'optimisations. À chaque itération, elle prend la meilleure solution parmi un groupe de solution candidats. Elle est plus rapide que la recherche exhaustive avec des résultats acceptables.

Dans notre cas, Nous avons développé une approche de glouton où l'on va faire à chaque itération :

- récupérer la liste des tous les coups possibles,
- évaluer chaque coup,
- appliquer le coup qui a la meilleure évaluation.

Avec cette approche, le paramètre le plus important pour obtenir de meilleurs résultats est l'efficacité de la fonction d'évaluation. En effet, puisque c'est elle qui va définir quel coup nous allons jouer à chaque itération, c'est donc elle qui va principalement influencer le résultat.

Dans le contexte du Flood-it, nous avons implémenté trois méthodes différentes pour l'évaluation de chaque coup.

La première méthode d'évaluation consiste à choisir la couleur qui permet de colorier le plus grand nombre de zones voisines. Pour cela, nous comptons combien il y a de zone dans le voisinage ayant la couleur correspondante au coup évalué.

La deuxième méthode d'évaluation consiste à choisir la couleur qui permet d'ajouter un nombre maximal de cellules dans le voisinage. Pour cela, nous récupérons les zones dans le voisinage ayant la couleur correspondante au coup évalué. Ensuite, nous comptons le nombre de cellules qui seront débloquées par ce dernier.

La troisième méthode d'évaluation consiste à choisir la couleur permettant de colorier la plus grande zone dans le voisinage.

L'exemple ci-dessous illustre le fonctionnement de la deuxième méthode d'évaluation sur une grille de 5x5. Les cases sélectionnées en orange désignent les cases choisies et les cases sélectionnées en bleu désignent les cases qui seront potentiellement accessibles.

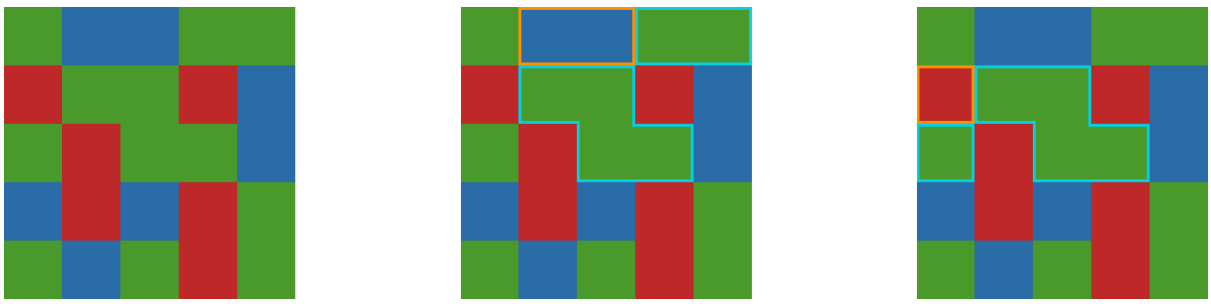


Figure 3.7 – Glouton - 1ère itération

Dans la figure 3.7 :
Le choix de la couleur “bleu” permet d’avoir accès à 6 cases.
Le choix de la couleur “rouge” permet d’avoir accès à 5 cases.
Alors nous allons colorer avec la couleur bleue.

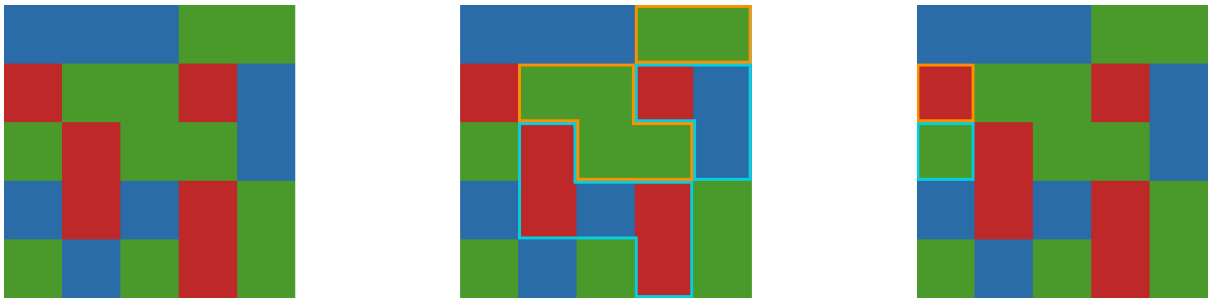


Figure 3.8 – Glouton - 2ème itération

Dans la figure 3.8 :
Le choix de la couleur “vert” permet d’avoir accès à 8 cases.
Le choix de la couleur “rouge” permet d’avoir accès à 1 case.
Alors nous allons colorer avec la couleur verte.

Figure 3.9 – Glouton - 3ème itération

Dans la figure 3.9 :
Le choix de la couleur “rouge” permet d’avoir accès à 6 cases.
Le choix de la couleur “bleu” permet d’avoir accès à 3 cases.
Alors nous allons colorer avec la couleur rouge.

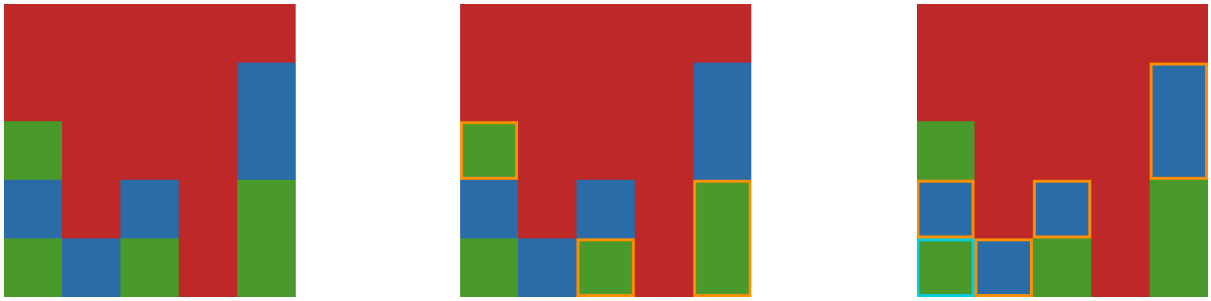


Figure 3.10 – Glouton - 4ème itération

Dans la figure 3.10 :
Le choix de la couleur "vert" permet d'avoir accès à 0 cases.
Le choix de la couleur "bleu" permet d'avoir accès à 1 cases.
Alors nous allons colorer avec la couleur bleue.

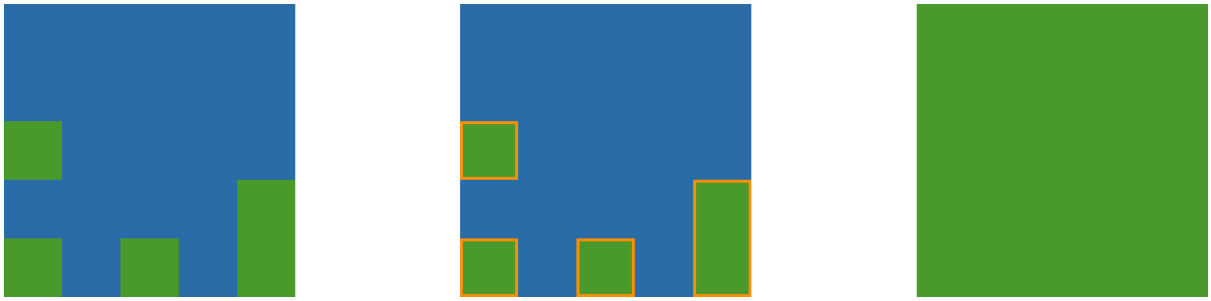


Figure 3.11 – Glouton - 5ème itération

Dans la figure 3.11 :

Il ne reste que la couleur "vert" donc nous sommes obligés de la choisir.

3.5 Recherche sur plusieurs coups

La recherche sur plusieurs coups est intermédiaire entre la recherche exhaustive et l'algorithme glouton. Le but étant d'essayer d'avoir de meilleurs résultats que l'algorithme glouton. En contrepartie, celle-ci prendra plus de temps.

L'approche adoptée dans la recherche sur plusieurs coups consiste à boucler entre l'approche utilisée dans la recherche exhaustive et l'approche utilisée dans l'algorithme glouton. Nous faisons exactement comme la recherche exhaustive, à la différence nous définissons une borne de K coups maximums (ce qui limite la taille de l'arbre de recherche à K). Enfin, nous évaluons l'état du jeu sur lequel nous avons effectué K coups. Une fois que toutes les combinaisons sont abouties, nous choisissons la combinaison avec le meilleur score et nous appliquons le premier coup. Nous refaisons cette procédure jusqu'à ce que le jeu soit résolu.

Comme dans l'algorithme glouton, la méthode d'évaluation influence le résultat final. En effet, c'est elle qui permet de déterminer la succession de coups pour une instance du jeu donnée. Pour cela, nous avons implémenté trois méthodes d'évaluation différentes.

Dans la première méthode d'évaluation, le but est de colorier le plus de cellules possibles. Pour cela, nous retournons le nombre de cases qui seront coloriées.

Dans la deuxième méthode d'évaluation, le but est de maximiser le nombre de cellules dans le voisinage. Pour cela, nous récupérons les zones dans le voisinage qui ont la couleur correspondante au coup évalué. Et pour chaque zone, nous faisons la somme du nombre de cellules qui sont dans celle-ci.

Dans la troisième méthode d'évaluation, le but est d'évaluer chacun des K coups d'une combinaison et de faire la somme de ces évaluations à la fin. Pour cela, nous avons utilisé la deuxième méthode d'évaluation de la recherche gloutonne qui permet d'évaluer un coup (car après quelques tests, cette méthode d'évaluation a donné de meilleurs résultats que les autres).

L'exemple ci-dessous illustre le fonctionnement de recherche sur plusieurs coups en utilisant la première méthode d'évaluation où K est fixé à 3 :

Nous avons les possibilités suivantes :
"rouge"→"bleu"→"vert" (score = 15).
"rouge"→"vert"→"bleu" (score = 18).
"rouge"→"vert"→"rouge" (score = 18).
"vert"→"bleu"→"rouge" (score = 20).
"vert"→"bleu"→"vert" (score = 12).
"vert"→"rouge"→"bleu" (score = 21).
"vert"→"rouge"→"vert" (score = 19).
nous choisissons la 7ème possibilité. Nous jouons donc : "vert".

Figure 3.12 – recherche sur plusieurs coups - 1ère itération

Nous avons les possibilités suivantes :
"bleu"→"rouge"→"bleu" (score = 22)
"bleu"→"rouge"→"vert" (score = 23)
"bleu"→"vert"→"rouge" (score = 22)
"rouge"→"bleu"→"rouge" (score = 22)
"rouge"→"bleu"→"vert" (score = 24)
"rouge"→"vert"→"bleu" (score = 24)
"rouge"→"vert"→"rouge" (score = 20)
on choisi la 5ème possibilité. Nous jouons donc : "rouge".

Figure 3.13 – recherche sur plusieurs coups - 2ème itération

Nous avons les possibilités suivantes :
“bleu”→“rouge”→“vert” (score = 25)
“bleu”→“vert”→“rouge” (score = 25)
“vert”→“bleu”→“rouge” (score = 25)
“vert”→“rouge”→“bleu” (score = 25)
on choisi la 1ère possibilité. Nous jouons donc : “bleu”.

Figure 3.14 – recherche su plusieurs coups - 3ème itération

Puisque toutes les cases restantes sont déjà accessibles et qu'il reste un nombre de coups inférieur à K, alors l'ordre de choix entre “vert” “rouge” n'a pas d'importance.

Figure 3.15 – recherche su plusieurs coups - 4ème et 5ème itération

3.6 Algorithme du plus court chemin

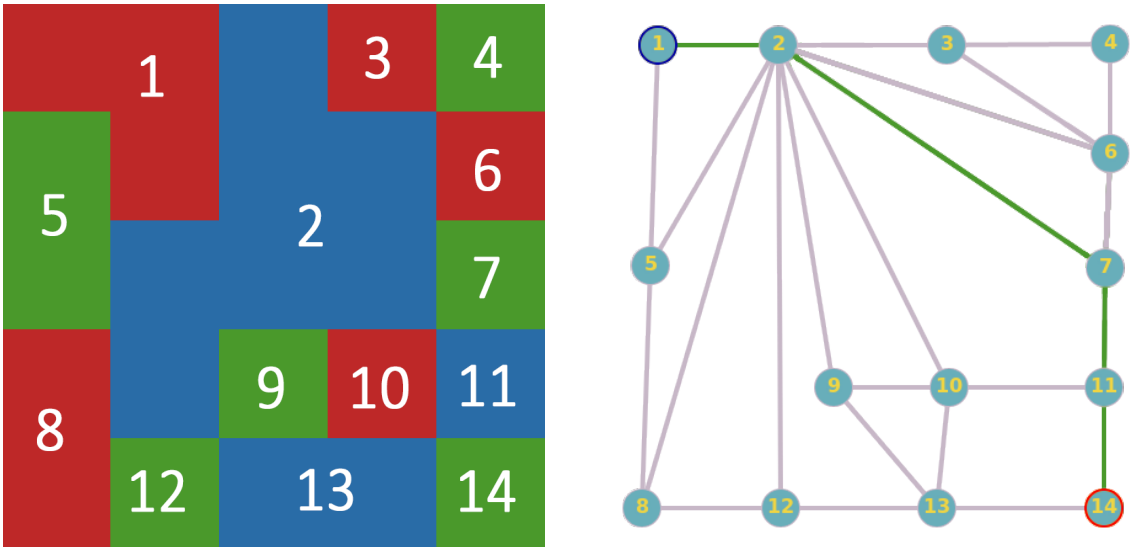
Dans l'algorithme du plus court chemin, nous avons représenté la grille par un graphe. Chaque sommet de ce graphe représente une zone différente. Un coup est représenté par une arrête où chacune d'entre elles à un poids fixe qui est égale à 1.

L'approche adoptée pour la recherche du plus court chemin consiste à répéter à chaque itération :

- Chercher la zone (le sommet) la plus éloignée de la zone initiale.
- Calculer le chemin le plus court pour atteindre cette zone.
- Colorier toutes les zones qui construit ce chemin

Dans cette approche, nous avons utilisé l'algorithme de Dijkstra [19] pour calculer le chemin le plus court

L'exemple ci-dessous illustre le fonctionnement de la recherche du plus court chemin. Pour Chaque figure nous représentons l'état du jeu et son graphe associé. Nous avons encerclé en bleu la zone initiale et en rouge la zone cible. Le chemin qui sera parcouru est en vert.



(a) État du jeu **(b)** Graphe représentatif

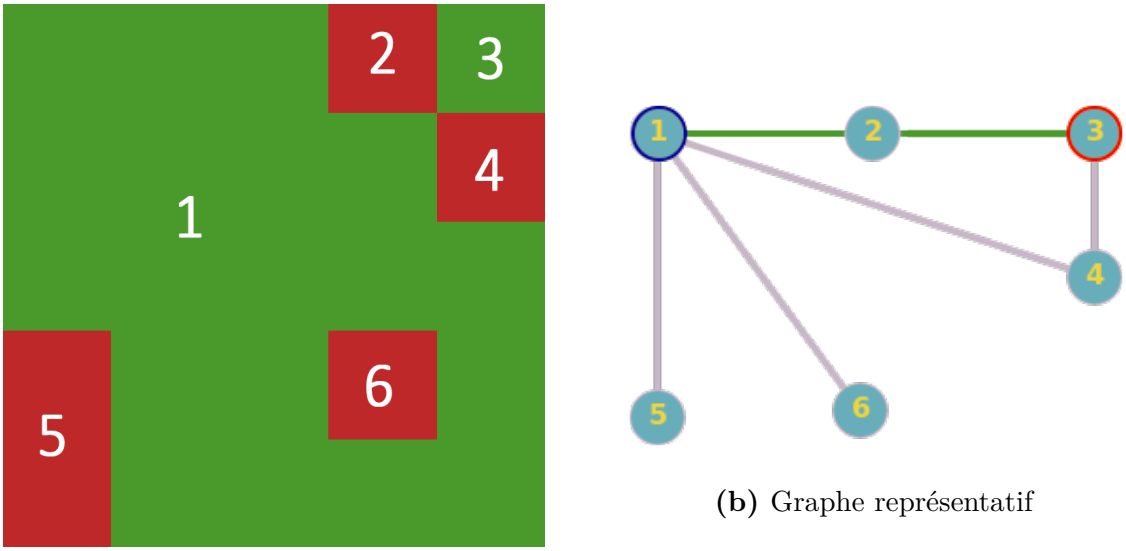
Figure 3.16 – Dijkstra - 1ère itération

Après avoir appliqué l'algorithme du plus court chemin sur le graphe 3.16b, nous avons obtenu les résultats suivants :

zone	parent	distance
2	1	1
3	2	2
4	3	3
5	1	1
6	2	2
7	2	2
8	2	2
9	2	2
10	2	2
11	7	3
12	2	2
13	9	3
14	11	4

Table 3.1 – Résultat de l'algorithme du plus court chemin - 1ère itération

À partir de ce tableau, nous distinguons bien que la zone 14 est la zone la plus distante de la zone initiale. Nous allons donc parcourir le chemin le plus court l'emmenant à celle-ci (il suffit de partir de la zone 14 en remontant à la zone 1 en suivant la colonne parent). Cela, donne la succession de coups suivants : "bleu"→"vert"→"bleu"→"vert".



(a) État du jeu

(b) Graphe représentatif

Figure 3.17 – Dijkstra - 1 ère itération

Après avoir appliqué l'algorithme du plus court chemin sur le graphe 3.17b, nous avons obtenu les résultats suivants :

zone	parent	distance
2	1	1
3	2	2
4	1	1
5	1	1
6	1	1

Table 3.2 – Résultat de l'algorithme du plus court chemin - 2 ème itération

À partir de ce tableau, nous distinguons bien que la zone 3 est la zone la plus distante de la zone initiale. Nous allons donc parcourir le chemin le plus court l'emmenant à celle-ci. Cela, donne la succession de coups suivants : "rouge"→"vert". Une fois cela effectué, la grille est donc résolue.

3.7 Recherche tabou

Lors de la conception nous n'avons pas réussi à définir le voisinage pour implémenter tabou. En effet, notre façon de définir le voisinage, nous aurait donné exactement les mêmes résultats que la recherche gloutonne. Par la suite, nous avons eu des contre-temps. Par conséquent nous n'avons pas réussi à implémenter cette partie qui était plus complexe que prévue.

Chapitre 4

Implémentation

4.1 Choix du langage de programmation

Nos encadrants pour ce projet ne nous ont pas imposé de langage de programmation. Par conséquent, nous étions libre d'en choisir un. Cependant, le choix d'un langage de programmation est quelque chose d'important et qui nécessite de réfléchir aux besoins et aux enjeux du projet. Dans notre cas, il y a plusieurs aspects à prendre en compte.

Un aspect important de notre projet est de pouvoir réutiliser les heuristiques sur d'autres jeux. Pour cela, nous avons modélisé une solution suivant un diagramme de classe. Le paradigme de programmation orienté objet est donc nécessaire. De plus, la possibilité d'utiliser le patron de conception "mixin" serait un atout, auquel cas nous devrions utiliser un modèle moins puissant.

Les heuristiques pouvant être coûteuses en temps et en espace, il nous faut un langage de programmation performant. De plus, la gestion de la mémoire et la possibilité de pouvoir intégrer des mécanismes d'optimisation seraient un atout.

Enfin, nous avions besoin de pouvoir implémenter une interface graphique assez simplement où la partie qui concerne l'affichage serait séparé du code.

Pour cela, nous avons comparé un panel de langages :

	Java	**C++**	**Python**	**PHP**
Paradigme objet	Oui	Oui	Oui	Oui
Héritage	Oui (héritage simple)	Oui (héritage multiple)	Oui (héritage multiple)	Oui (héritage simple)
Classe abstraite et Interface	Oui	Oui	Oui	Oui
Généricité	Oui (mais usage assez restrictif)	Oui (duck typing [10])	Non (pas de typage)	Non (pas de typage)
Héritage paramétré (mixin)	Non	Oui	Non	Non
Gestion de la mémoire	Garbage collector [11]	Géré par le développeur	Garbage collector [11]	Garbage collector [11]
Performance	Performant	Très performant	Peu performant	Peu performant
Concevoir une interface graphique	Facile	Facile	Facile	Très facile

Table 4.1 – Tableau comparatif des langages de programmation

C'est donc tout naturellement que nous avons choisi C++. De plus, n'ayant pas pu utilisé tout le potentiel de ce langage durant nos années à la Faculté, c'était l'occasion pour nous de voir la puissance que peut offrir ce langage. En effet, C++ offre de nombreuses possibilités sur l'utilisation du paradigme objet (qui évolue au fur et à mesure des nouvelles spécifications) et une librairie complète. Grâce à certains mécanismes que propose le langage, nous pouvons aussi intégrer pas mal d'optimisation car C++ est un langage compilé. Cela a donc nécessité un travail de recherche important pour utiliser au mieux ce langage.

4.2 Développement d'une interface graphique

En ce qui concerne l'interface graphique, nous avons choisi d'utiliser l'IDE Qt Creator. Ce dernier est un environnement de développement intégré multi-plateforme faisant partie du framework Qt. Il permet de pouvoir concevoir facilement des interfaces graphiques en C++ (il est aussi possible de développer ces interfaces en Python avec cet outil). En effet, grâce à ce framework, la partie visuelle et la partie code sont séparées (la description de la vue se fait dans un fichier xml). Cela permet une simplification et une meilleure réutilisation du code. De plus, grâce à ce système, Qt propose l'outil Designer permettant de construire facilement des interfaces graphiques. Cela permet d'éviter d'écrire le code XML décrivant l'interface graphique

(ce qui est assez fastidieux) et de gagner du temps (voir Annexe A.1).

Voici l'interface graphique que nous avons réalisée :

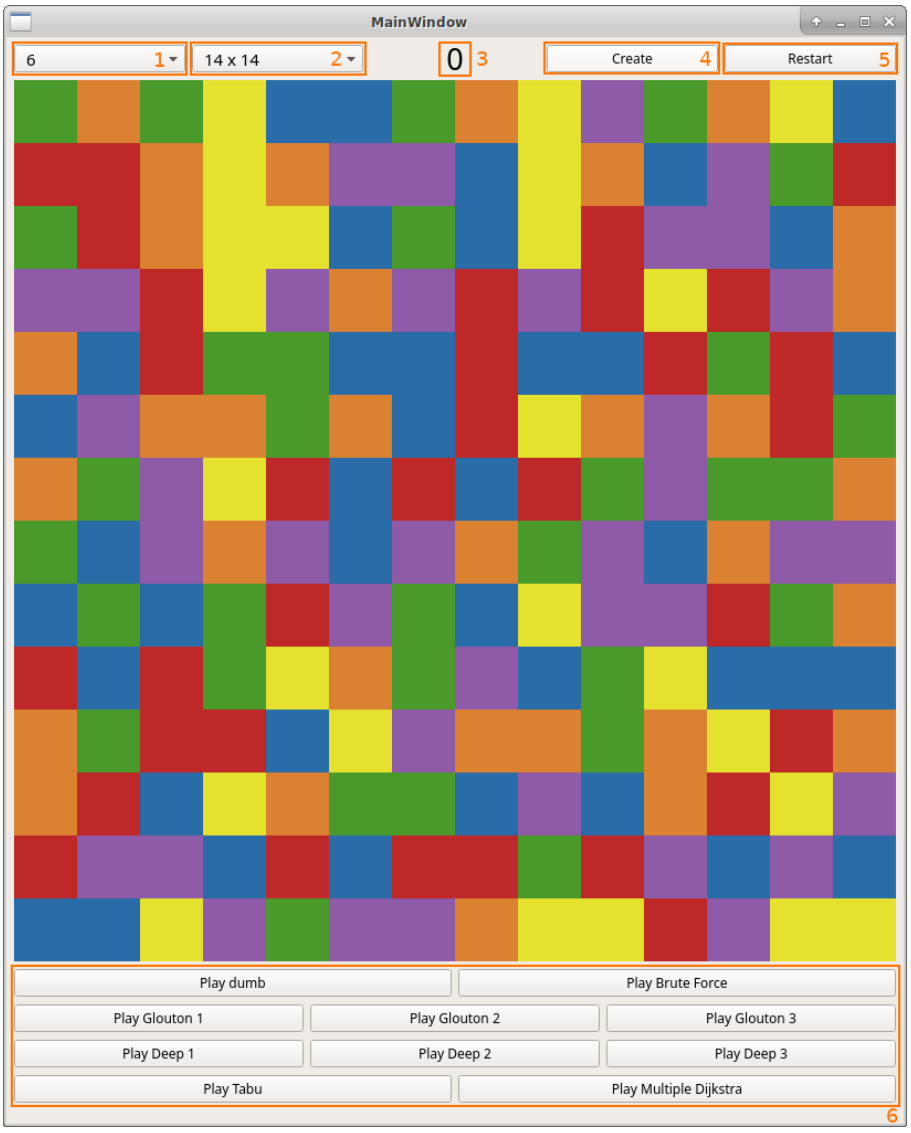


Figure 4.1 – Interface graphique

— 1 : choix du nombre de couleur,

— 2 : taille de la grille,

— 3 : nombre de coups effectués,

— 4 : créer une nouvelle grille aléatoire,

— 5 : recommencer la partie courante,

— 6 : appliquer une méthode de résolution (les différents boutons d'une même heuristique sont des variantes où la méthode d'évaluation est différente) : cela lancera une animation pour visualiser la résolution coup à coup.

4.3 Gestion de la mémoire et C++ moderne

Un point important mentionné précédemment, c'est que C++ n'a pas de ramasse-miette (Garbage Collector [11]). Cela signifie que la mémoire doit être gérée par le développeur. Par conséquent, les possibilités d'optimisations sont plus nombreuses. En effet, les ramasses-miettes consomment des ressources en mémoire importantes et sont facteurs de ralentissement. Cependant, ils garantissent qu'il n'y ait pas de fuite de mémoire [12], alors que la gestion manuelle de la mémoire est un facteur de risque. Il n'y a aucune garantie sur la bonne gestion de celle-ci.

Pour remédier aux problèmes que peuvent induire la gestion manuelle de la mémoire, il existe ce que l'on appelle des pointeurs intelligents (Smart Pointer) [13]. Ils ont été introduits depuis le standard C++11. Cela permet de pouvoir quasiment assurer qu'il n'y aura pas de fuite de mémoire. En effet, les pointeurs sont encapsulés dans un objet donnant accès à différentes fonctionnalités dessus. Pour cela, il existe 4 types de pointeurs :

— Unique pointer,

— Shared pointer,

— Weak pointer.

Un unique pointeur permet d'assurer qu'il est l'unique détenteur d'un pointeur. Par conséquent, lorsque celui-ci est supprimé, il libère le pointeur qu'il détient. Pour cela, nous pouvons faire diverses opérations :

— déplacer le pointeur encapsulé vers un autre *unique pointer*,

— remplacer le pointeur courant (qui sera donc libéré) par un autre,

— directement appelé les méthodes de l'objet,

— déréférencer le pointeur.

De plus, la copie d'un *unique pointer* est impossible car cela signifierait qu'il existe 2 *unique pointer* partageant un même pointeur.

Par conséquent, pour assurer la bonne gestion de la mémoire, le développeur doit s'assurer qu'il n'y a pas des *unique pointer* qui partagent un même pointeur. Pour cela, il faut distinguer qui est le propriétaire de l'objet puisque c'est lui qui se chargera de l'allocation et la désallocation de celui-ci. Les autres devront avoir une référence vers l'objet (nous pouvons voir une référence comme un pointeur non nul sur lequel on ne peut pas libérer l'objet en mémoire).

Les *shared pointer* et les *weak pointer* peuvent partager un même pointeur entre eux, contrairement aux *unique pointer*. Ces 2 types de pointeurs fonctionnent ensemble : les *shared pointer* sont responsables de la désallocation de la mémoire, les *weak pointer* non. Ils ont les mêmes fonctionnalités qu'un unique pointeur à la différence qu'on peut les copier. Pour chaque pointeur on compte le nombre de fois qu'ils sont partagés entre différents *shared pointeur*. Ainsi, lorsque le compteur arrive à 0 pour un pointeur, il sera désalloué. Par conséquent, le développeur doit bien comprendre son modèle afin de distinguer qui doit être responsable de la libération de l'objet en mémoire et qui ne l'est pas.

Pour utiliser les smart pointer, il est important de bien concevoir et comprendre son modèle afin de les utiliser au mieux. En effet, il est préférable d'éviter au maximum d'utiliser les *shared*

pointer car ils sont plus compliqués à mettre oeuvre. Pour cela, il est préférable d'utiliser les *unique pointer* à la place des *shared pointer* lorsque cela est possible. De plus, il est beaucoup plus facile de remplacer des *unique pointer* par des *shared pointer* alors que l'inverse peut s'avérer être une tâche très compliquée.

La communauté de développeur C++ conseille fortement l'utilisation de ces smart pointer. Nous nous sommes donc posés la question de l'impact sur les performances qu'ils peuvent engendrer. Pour cela, nous avons effectué 2 benchmark avec l'utilisation des pointeurs simples, *unique pointer* et des *shared pointer*. Le premier sans optimisation faite à la compilation et le deuxième avec. Voir le code à l'annexe A.2. Voici le résultat obtenu :

```
g-eremy@geremy-VirtualBox:~/Documents/test$ g++ -o t3 t3.cpp
g-eremy@geremy-VirtualBox:~/Documents/test$ ./t3
---------------------------
Test sans optimisation à la compilation
---------------------------
create raw pointer : 214273µs
create unique pointer : 300897µs
create shared pointer : 399753µs
---------------------------
delete raw pointer : 85426µs
delete unique pointer : 168511µs
delete shared pointer : 258824µs
---------------------------
g-eremy@geremy-VirtualBox:~/Documents/test$ g++ -o t3 t3.cpp -O3
g-eremy@geremy-VirtualBox:~/Documents/test$ ./t3
---------------------------
Test avec optimisation à la compilation
---------------------------
create raw pointer : 132078µs
create unique pointer : 132564µs
create shared pointer : 253402µs
---------------------------
delete raw pointer : 48382µs
delete unique pointer : 41799µs
delete shared pointer : 108559µs
---------------------------
g-eremy@geremy-VirtualBox:~/Documents/test$
```

Figure 4.2 – Benchmark smart pointer

Nous constatons que les *unique pointeur* ont un impact négligeable sur les performances, alors que les *shared pointeur* ont un impact significatif. Par conséquent, nous avons décidé d'utiliser uniquement des *unique pointer* dans notre projet afin d'avoir les meilleures performances possible et une bonne gestion de la mémoire. Pour cela, nous avons dû analyser notre modèle pour définir quelles instances seront propriétaires des différents objets, et lesquelles détiendront des références vers ceux-ci.

Pour finir, nous voulions utiliser au maximum la puissance du langage C++. Pour cela, nous nous sommes documentés sur les optimisations pouvant être effectuées à la compilation qu'il existe dans ce langage. Nous avons trouvé de nombreuses ressources. Par exemple, il y a le déplacement d'objet (avec le constructeur par déplacement), la déclaration des fonctions "inline", la "copy elision", etc... L'utilisation de ce langage a donc été bénéfique pour notre projet et notre apprentissage.

Chapitre 5

Conduite de projet

5.1 Planning

Au début du projet, nous avions dressé un planning prévisionnel pour situer dans le temps l'avancement du projet et de chaque tâche. À la fin du projet, nous pouvons remarquer qu'il y a quelques différences entre ce que nous avions prévu et ce qui s'est réellement passé.

Pour marquer cette différence, nous avons dressé un planning final :

5.1.1 Planning prévisionnel

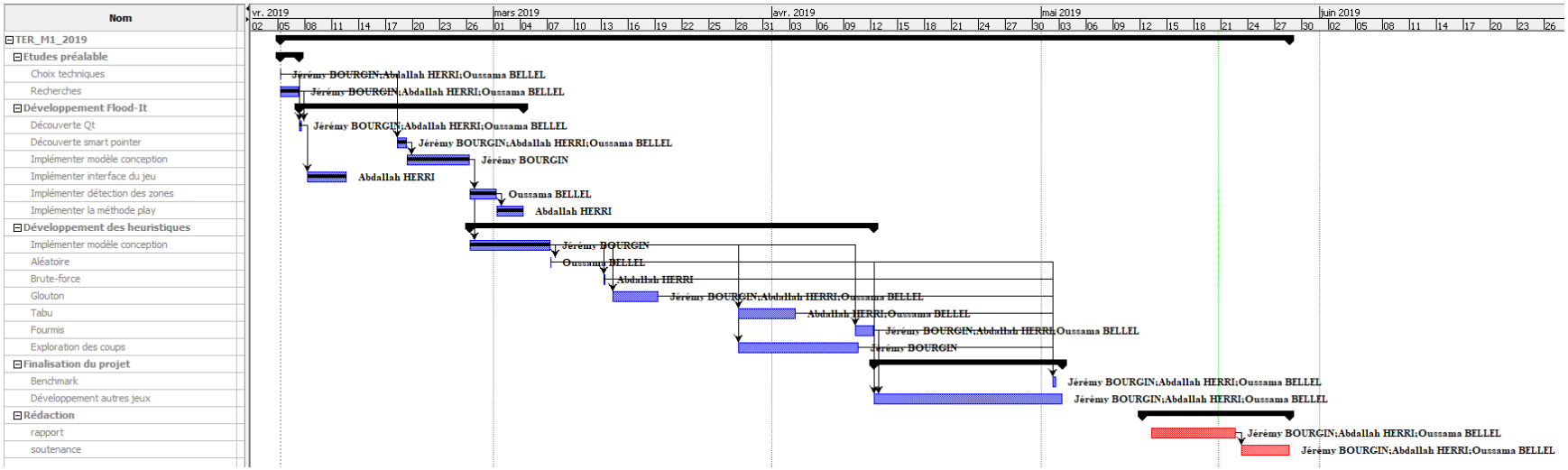


Figure 5.1 – Planning prévisionnel - Diagramme de Gantt

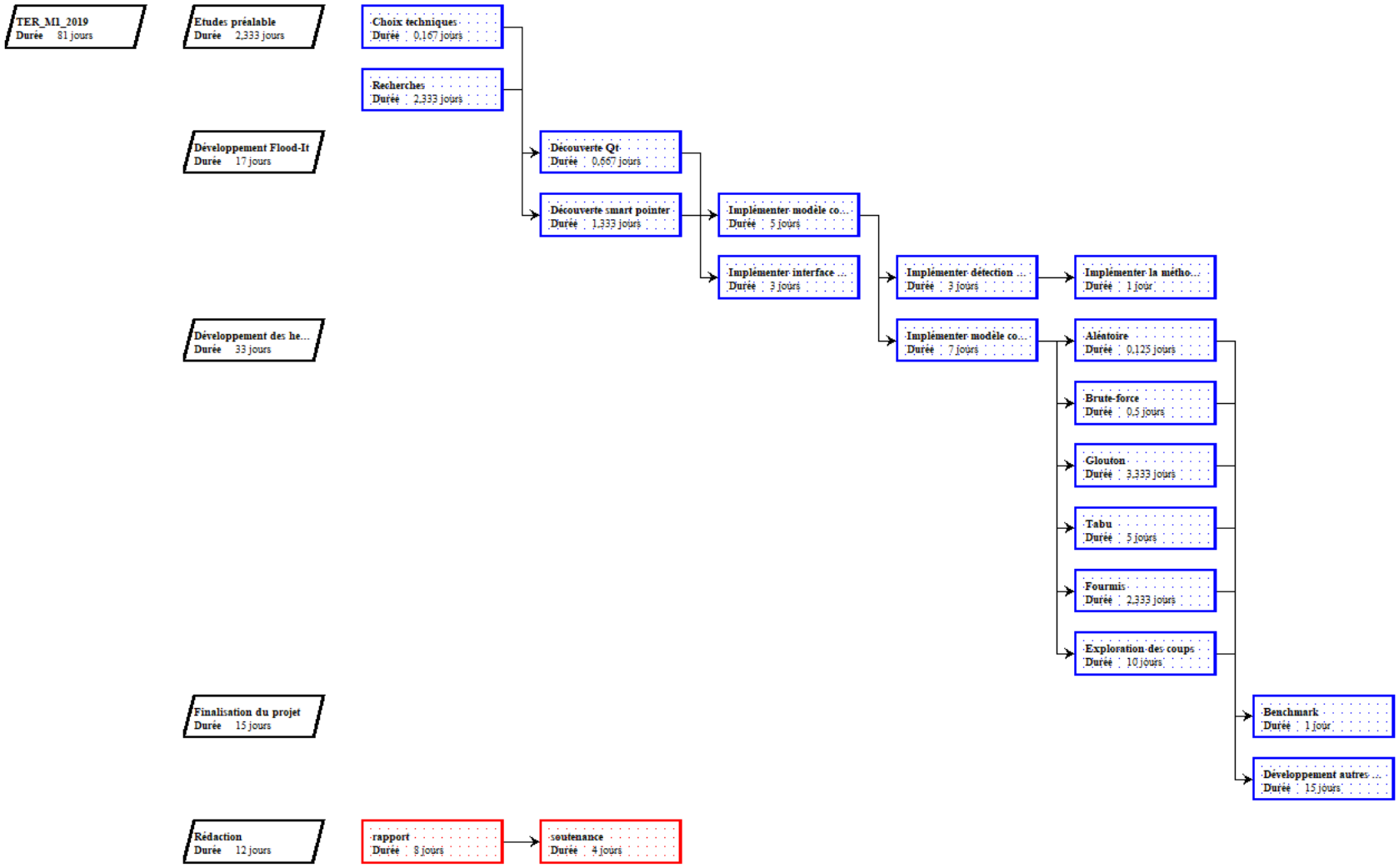


Figure 5.2 – Planning prévisionnel - Diagramme de Pert

5.1.2 Planning final

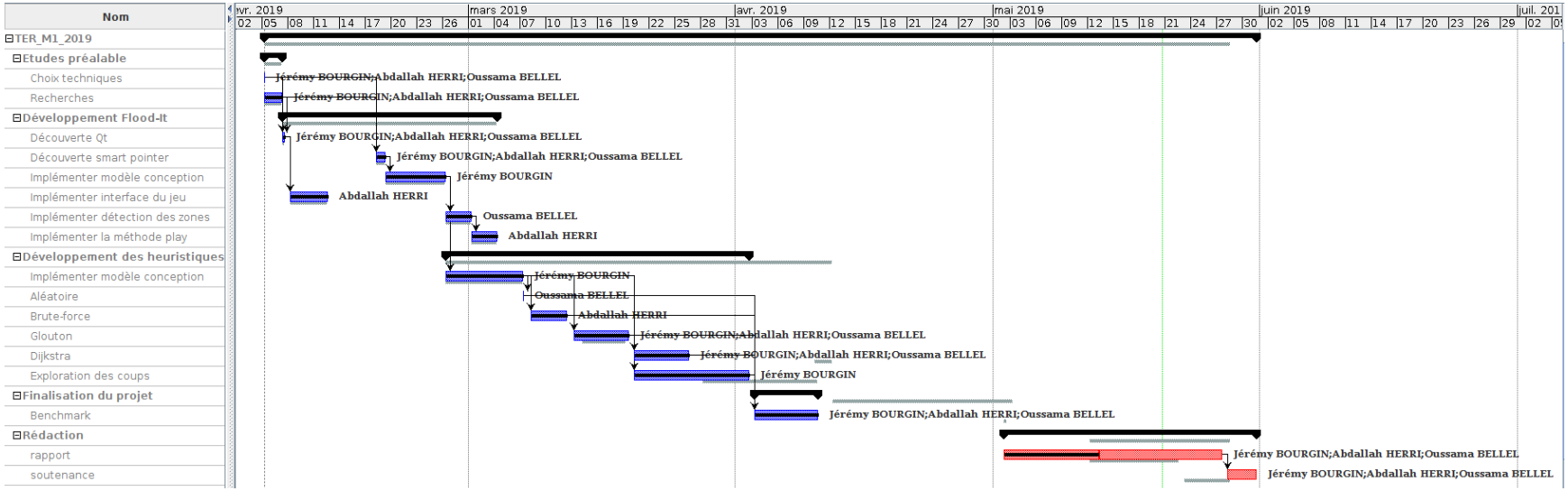


Figure 5.3 – Planning final - Diagramme de Gantt

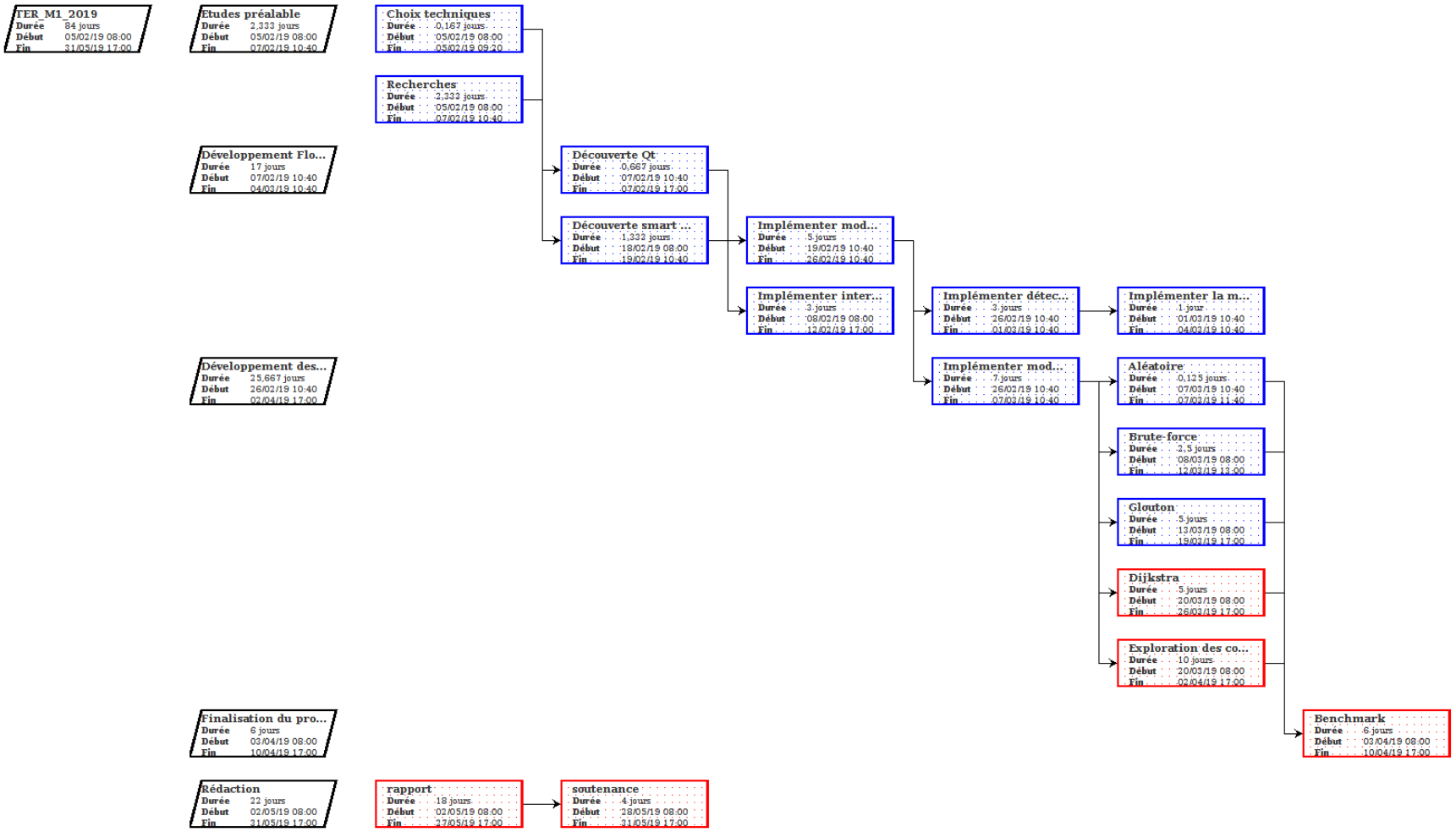


Figure 5.4 – Planning final - Diagramme de Pert

5.2 Déroulement du projet

5.2.1 Données quantitatives

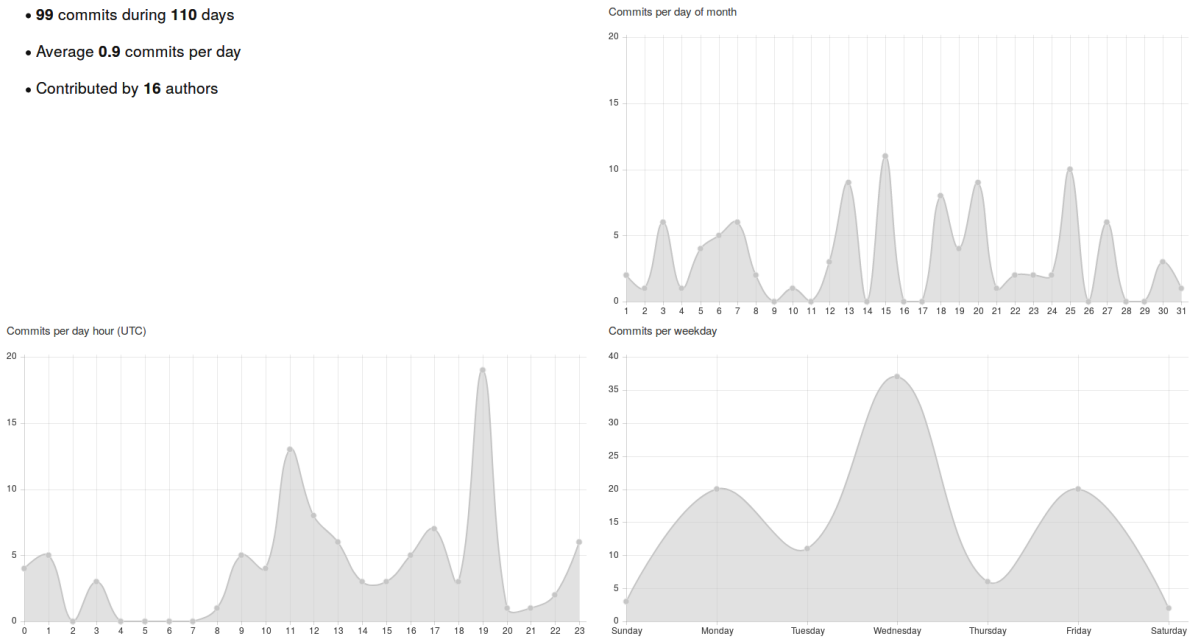


Figure 5.5 – Graphes des commits

La charge de travail de notre projet se répartit essentiellement sur le Mardi, Mercredi et Vendredi. En effet, chaque Mardi nous avions rendez-vous par nos encadrants de TER. Par conséquent, cela donnait lieu à quelques modifications. Enfin, le Mercredi et le Vendredi, nous n'avions pas cours, c'était donc pendant ces 2 jours que nous passions la plupart du temps de travail sur le projet.

À l'issue de notre projet, voici la liste des principaux éléments quantitatifs que nous avons réalisés :

- lien vers le GIT (site) : https ://gitlab.info-ufr.univ-montp2.fr/e20140034048/TERM1,
- lien vers le GIT (clone) : git@gitlab.info-ufr.univ-montp2.fr :e20140034048/TERM1.git,
- environ 200h de travail pour chaque membre du groupe (sachant que nous avons travaillés environ 10h le mercredi et le vendredi pendant 2 mois),
- nombre de classes : 28 classes et 8 interfaces (total de 74 fichiers),
- 3200 lignes de codes,
- pas de bugs / problèmes connus à ce jour,
- features effectuées :
  - flood-It + interface graphique,
  - modèle de conception réutilisable,
  - méthode de résolution exacte (brute-force),
  - méthode de résolution naïve (recherche aléatoire),
  - l'heuristique gloutone (avec plusieurs versions),
  - l'algorithme Dijkstra,
  - recherche sur plusieurs coups,
  - benchmark des différentes heuristiques
- features non effectuées :
  - implémenter la recherche tabou,
  - feature bonus : implémenter le démineur + fonction d'évaluation de coup
- features à améliorer :
  - optimiser brute force & la recherche su plusieurs coups (multi-threading),
  - optimiser l'animation de résolution

5.2.2 Données qualitatives

Ici, nous allons analyser les différences entre le planning prévisionnel et le planning final. Pour cela, nous allons commencer par expliquer différences entre les deux, puis nous allons en expliquer les causes. Cela permettra de mieux comprendre comment nous avons géré notre projet.

Nous avons changé la répartition des tâches et leur durée dans le temps. Pour la majorité des tâches dans la partie "développement des heuristiques" et "finalisation du projet", nous avons dû leur attribuer plus de temps. En effet, notre méthodologie de travail faisait qu'une tâche durait au moins une semaine. Elles devaient être validées avec nos encadrants chaque semaine. Cependant, nous avons mieux organisé et réparti notre temps de travail. Ce qui explique que nous avons quand même pu finir ces tâches plus rapidement que prévu.

Nous n'avons pas pu travailler sur le projet pendant une période importante. Cela explique pourquoi il y a des tâches qui ont été supprimées :

— tabu,

— développement d'autres jeux

et d'autres tâches ont été remplacées par d'autres :

— Fourmis par Dijkstra

Pour les tâches qui ont subi des changements, il y a "Tabu" qui a été supprimé, car une fois la conception commencé nous avons réalisé qu'elle était plus compliquée que prévu. Quant à "Implémentation d'autres jeux" nous l'avons supprimé, car nous n'avons pas eu le temps. De plus ce n'était pas une tâche importante puisque nous n'étions pas obligé de la faire. Pour la tâche "Fourmis", nous avons décidé de la remplacer par "Dijkstra" car elle était plus adaptée dans notre contexte.

Pendant le mois d'avril, nous avons eu une période creuse, cela s'explique par le fait que nous avions eu d'autres projets qui sont arrivés tardivement et que nous devions finir rapidement. Pour anticiper le risque d'avoir une période pendant laquelle nous ne pourrons pas ou peu consacrer du temps sur ce projet, nous avons décidé d'y dédier plus de temps. Par conséquent, nous n'avons pas pu finir la tâche "Tabu" car nous nous sommes retrouvés bloqués à l'étape de conception pendant cette période. Cela explique pourquoi nous n'avons pas réussi à concrétiser cette tâche. Cependant, l'anticipation de ce risque nous a permis de pouvoir à boucler les tâches les plus importantes du projet.

5.3 Organisation du travail

Pour conclure cette partie, nous avons remarqué que nous avons suivi le cycle de vie en spirale (voir annexe A.3) comme méthodologie de travail pour notre projet. En effet, tous les mardis nous avions rendez-vous avec nos encadrants. Lors de ces réunions, nous faisions la spécification, l'analyse et la validation de chaque fonctionnalité. Le processus de validation se faisait lorsque nous avions fini d'implémenter une fonctionnalité, afin de s'assurer que nous n'avions pas fait d'erreur (cela est arrivée une seule fois avec l'algorithme d'exploration sur plusieurs coups). Nous faisions la conception, l'implémentation et les tests des fonctionnalités lors de nos séances de travail (donc principalement le mercredi et le vendredi). Cette méthodologie de travail nous a permis de pouvoir implémenter les fonctionnalités les plus importantes et éviter des erreurs au niveau de la spécification de celle-ci.

De plus, nous avons passé beaucoup de temps sur la conception d'un modèle permettant de rendre réutilisable nos heuristiques, qu'elles soient indépendantes entre elles, que le jeu, les heuristiques ainsi que l'interface graphique soient tous autonomes. Ainsi, cela nous a permis de pouvoir utiliser la méthodologie de travail du cycle en spirale (puisque la modification et l'ajout d'une heuristique n'entraînent aucune conséquence sur les autres) et d'éviter tout ce qui peut être source d'erreur dès le début du projet.

Pour finir, la gestion de ce projet est dans sa globalité une réussite.

Chapitre 6

Résultats

6.1 Observations

Après avoir implémenter et tester le fonctionnement de plusieurs heuristiques, nous souhaitions analyser et comparer celles-ci. Dès lors, nous avons voulu rendre leurs résolutions animées (donc voir ce que font les heuristiques coup après coup) via l'interface graphique (voir figure 4.1). Ainsi, nous pouvions voir le comportement de chacune d'entre-elles.

Après avoir effectué plusieurs tests, nous avions remarqué que sur des grilles de taille assez grande, nous pouvions voir le comportement différer d'une heuristique à l'autre. Pour cela, nous allons les comparer sur une grille de taille 100x100 comportant 6 couleurs.

6.1.1 Algorithme glouton

Après avoir faire plusieurs tests, nous avons observé que glouton se propage rapidement sur toute la grille. Peu importe les méthodes d'évaluation utilisées, nous observons toujours le même comportement. Voici un exemple de l'application de l'algorithme pour appuyer nos propos :

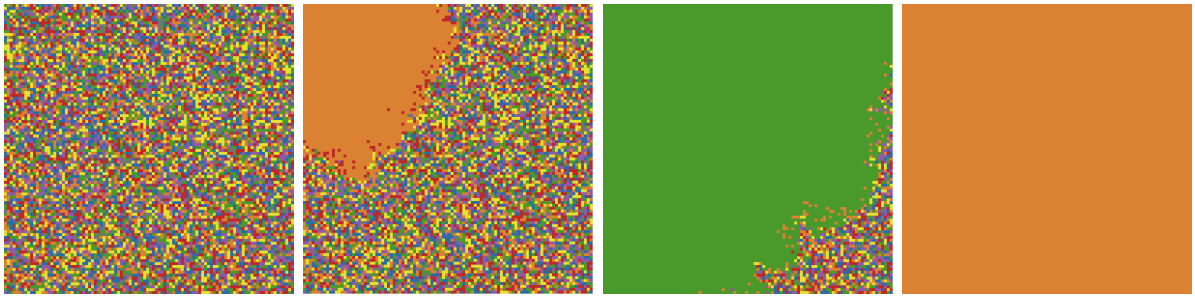


Figure 6.1 – Glouton - Progression

6.1.2 Recherche su plusieurs coups

Dans le cas de l'algorithme qui fait la recherche sur plusieurs coups, nous avons constaté des différences de comportement en fonction des méthodes d'évaluation utilisées.

Avec la première et la troisième méthode d'évaluation, l'heuristique a un comportement semblable à celui de l'algorithme glouton. Cela s'explique par le faite que leur façon de fonctionner ont des points en commun. La figure 6.2 ci-dessous nous montre un exemple d'exécution de cet algorithme en utilisant la troisième méthode d'évaluation :

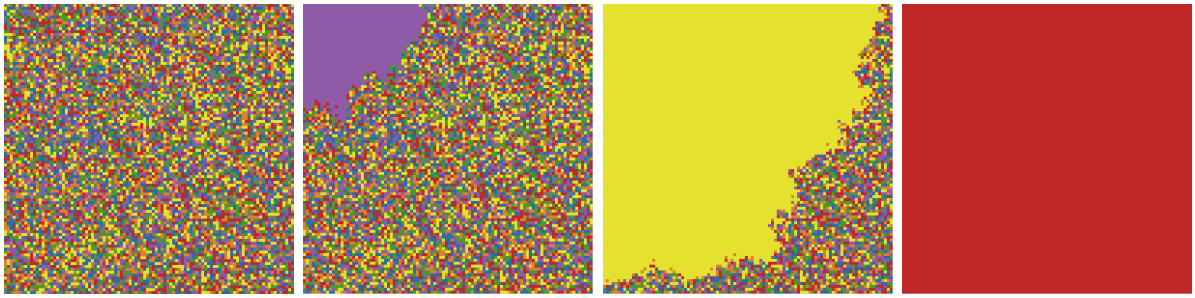


Figure 6.2 – Recherche sur plusieurs coup avec la 3ème méthode d'évaluation - Progression

Après avoir testé avec la deuxième méthode d'évaluation, nous avons remarqué un comportement inattendu. L'algorithme a tendance à négliger une ou plusieurs couleurs. Cela se produit quand le choix de cette couleur diminue le nombre de zones qu'il y a dans le voisinage. Par conséquent, l'algorithme ne choisira pas la couleur en question, sauf dans le cas où elle sera contrainte de l'utiliser (donc quand le choix des coups possibles est réduit à cette couleur). Pour appuyer nos propos, l'exemple ci-dessous est l'application de l'algorithme avec cette méthode d'évaluation :

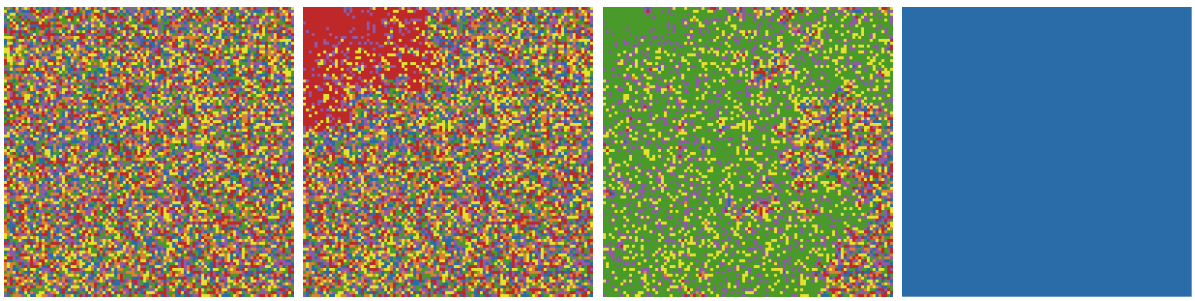


Figure 6.3 – Recherche sur plusieurs coup avec la 2ème méthode d'évaluation - Progression

6.1.3 Recherche du plus court chemin

La figure 6.4 ci-dessous nous montre le résultat d'application de l'algorithme de recherche du plus court chemin. Nous remarquons qu'à chaque itération, il "creuse" très rapidement vers un point précis de la grille. Dans ce cas, ce comportement était celui attendu.

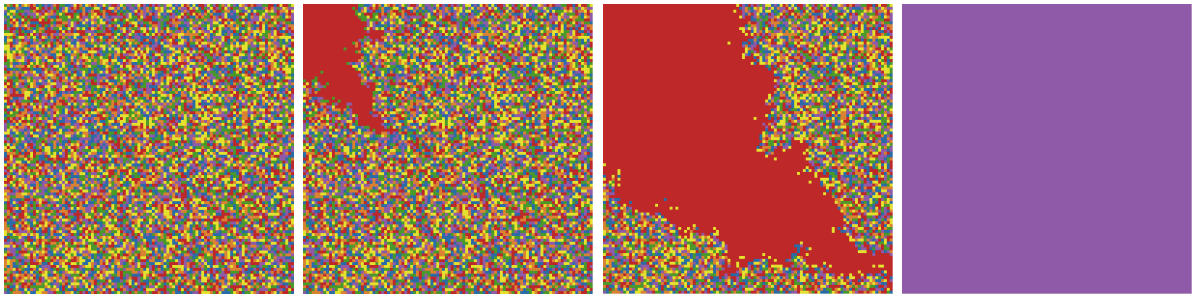


Figure 6.4 – Recherche du plus court chemin - Progression

6.2 Statistiques

Une fois que nous avions observé le comportement de chaque heuristique, nous voulions cette fois ci comparer leur résultat. En effet, il est primordial de savoir quelles heuristiques résolvent le jeu en effectuant le moins de coups. De plus, nous souhaitions savoir quelles étaient leur efficacité en cherchant le nombre de coup et le temps effectué pour une résolution. Il fallait donc faire un benchmark.

Pour cela, nous avons réalisé une série de mesures avec un processus bien précis. En effet, si l'on souhaite obtenir des résultats permettant de pouvoir comparer les différentes heuristiques, il est nécessaire que le processus soit établi rigoureusement.

Nous avons donc créé un jeu de données de 1080 grilles comportant :

- 10 grilles de taille 5x5 dans un intervalle de 3 à 10 couleurs
- 10 grilles de taille 6x6 dans un intervalle de 3 à 10 couleurs
- 10 grilles de taille 7x7 dans un intervalle de 3 à 10 couleurs
- 10 grilles de taille 8x8 dans un intervalle de 3 à 10 couleurs
- 10 grilles de taille 9x9 dans un intervalle de 3 à 10 couleurs
- 10 grilles de taille 10x10 dans un intervalle de 3 à 10 couleurs
- 10 grilles de taille 15x15 dans un intervalle de 3 à 10 couleurs
- 10 grilles de taille 20x20 dans un intervalle de 3 à 10 couleurs
- 10 grilles de taille 50x50 dans un intervalle de 5 à 20 couleurs
- 5 grilles de taille 100x100 dans un intervalle de 5 à 30 couleurs
- 5 grilles de taille 200x200 dans un intervalle de 5 à 30 couleurs
- 1 grille de taille 500x500 dans un intervalle de 5 à 15 couleurs
- 1 grille de taille 1000x1000 dans un intervalle de 5 à 15 couleurs

Par conséquent, nous pourrons comparer chaque heuristique car elles auront travaillé sur les même grilles. De plus, nous avons réduit considérablement le facteur aléatoire en générant plusieurs instances de grilles pour une taille et une couleur donnée (sauf dans le cas où les grilles sont très grandes car la complexité est telle que le facteur aléatoire est déjà très faible). Ensuite, nous avons appliqué chaque méthode de résolution sur l'ensemble du jeu de données.

Ainsi, nous avons pu savoir en combien de coups et de temps une méthode de résolution permet de résoudre une grille du jeu de données. Cependant, puisqu'il y avait des grilles qui pouvaient être plus complexes que d'autres, la recherche exhaustive et la recherche sur plusieurs coups pouvaient prendre trop de temps. Nous avons donc décidé de les appliquer sur des instances raisonnables.

À l'issue du benchmark, nous avons pu récupérer les résultats dans un fichier, et les stocker dans une base de données (voir Annexe A.4). Ainsi, il était plus facile de pouvoir établir des statistiques sur des portions de données. À cet effet, nous avons groupé les résultats uniquement sur les tailles des différentes grilles. Puis, nous avons calculé la moyenne et l'écart type du nombre de coups et le temps qu'effectue chaque méthode de résolution (voir Annexe A.5).

La légende ci-dessous désigne la couleur attribuée à chacune des courbes. Cette couleur sera fixe pour tous les graphes qui suivent :

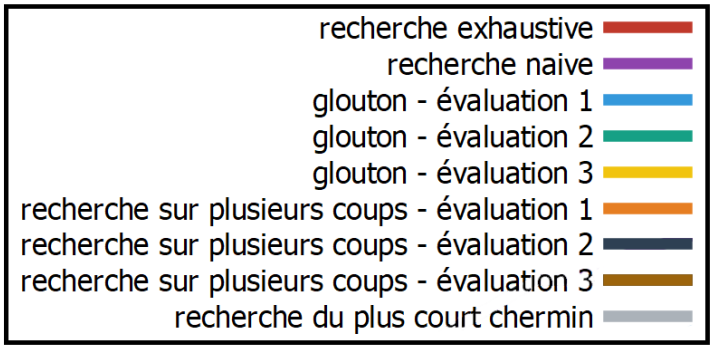


Figure 6.5 – Graphe de comparaison - Légende

Nous allons donc commencer par comparer le résultat optimal (donc les résultats issus de la recherche exhaustive) avec le résultat que donnent nos méthodes de résolution :

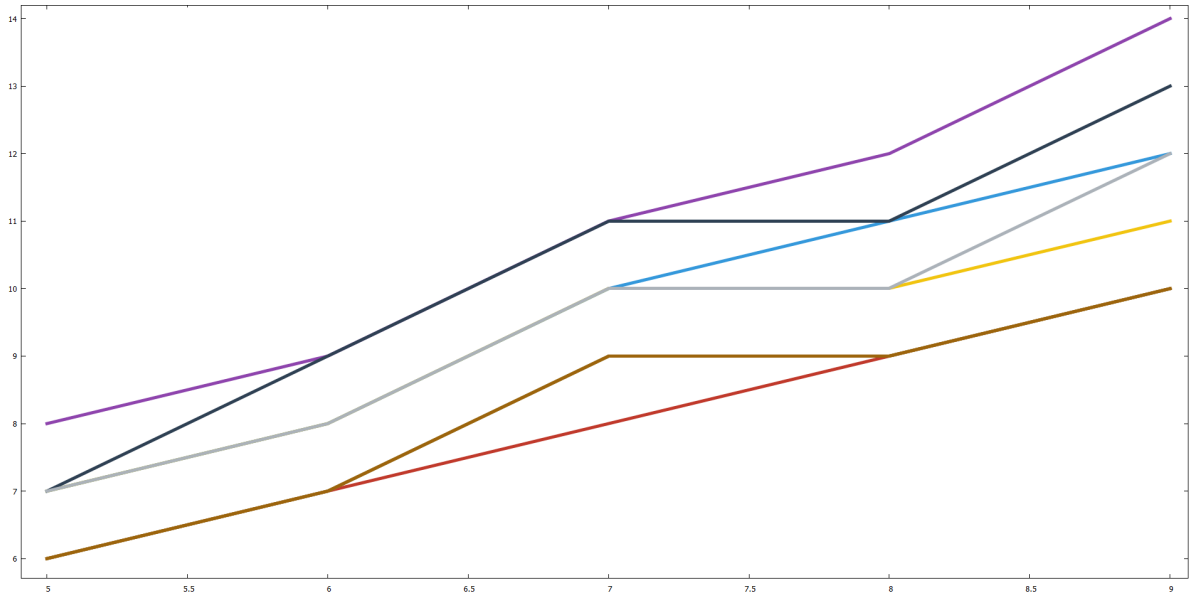


Figure 6.6 – Graphe de comparaison 1 - Nombre de coups moyen

Dans le graphe 6.6 ci-dessus, nous avons mesuré le nombre de coups moyen effectué par chaque méthode de résolution en fonction de la taille de la grille. Nous remarquons que tous les algorithmes sont proches de l'optimal. Cependant, la recherche sur plusieurs coups avec la deuxième méthode d'évaluation est plus proche de la recherche naïve. Donc à l'exception de cette méthode de résolution, les algorithmes semblent donner de bonnes approches. À ce stade, nous ne pouvons pas donner d'affirmation sur :

— la tendance de la courbe optimale,

— la taux de dérivation de chaque heuristique par rapport à l'optimal,

— dire qu'une heuristique est meilleure que l'autre

En effet, nous avons appliqué la recherche exhaustive sur une plage très restreinte.

Nous souhaitons maintenant comparer les heuristiques entre elles. Pour cela, nous allons commencer par comparer la recherche gloutonne, et la recherche sur plusieurs coups puisque ces 2 méthodes sont plus ou moins proche :

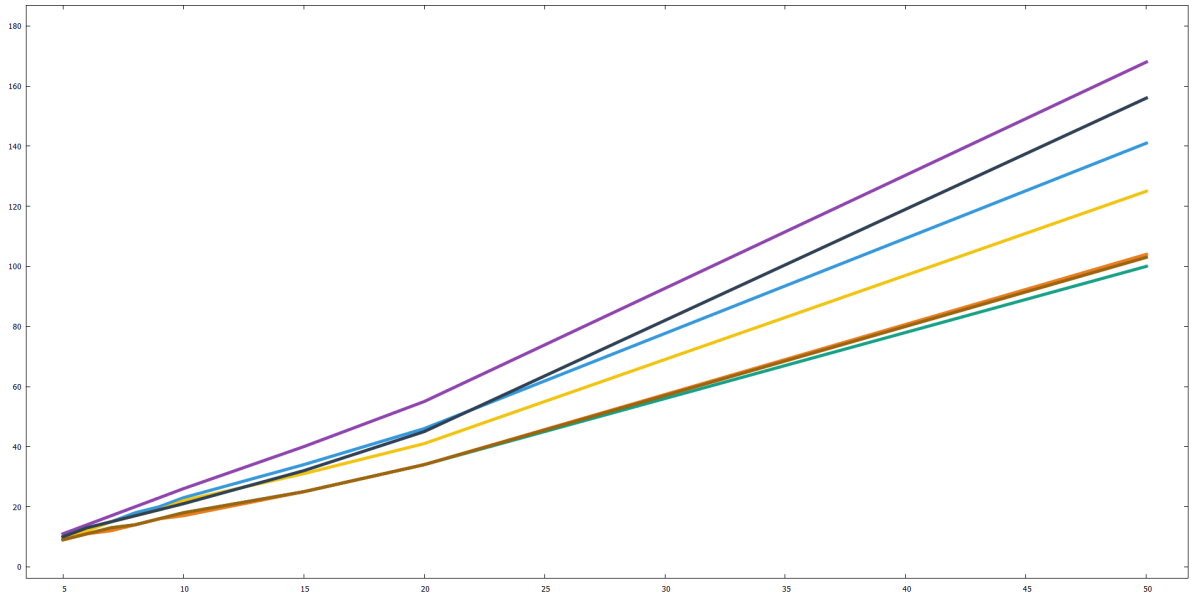


Figure 6.7 – Graphe de comparaison 2 - Nombre de coups moyen

Dans le graphe 6.7 ci-dessus, nous avons mesuré le nombre de coups moyen en fonction de la taille de la grille.

Nous remarquons que les méthodes de résolutions suivantes donnent des résultats très proches :

- recherche sur plusieurs coups - première méthode d'évaluation,
- recherche sur plusieurs coups - troisième méthode d'évaluation,
- recherche gloutonne - deuxième méthode d'évaluation

Nous remarquons aussi que ces méthodes donnent de bons résultats. En effet, elles évoluent moins vite que les autres, et sont très éloignées de la courbe de la recherche naïve. Les deux autres recherches gloutonnes donnent des résultats qui sont acceptables. Cependant, nous remarquons que la recherche sur plusieurs coups avec la deuxième méthode d'évaluation donne de mauvais résultats. Effectivement, son évolution est mauvaise, et elle tend à se rapprocher, voir dépasser la recherche naïve. Par conséquent, essayer de garder une ou deux couleurs jusqu'à la fin semble être une mauvaise idée.

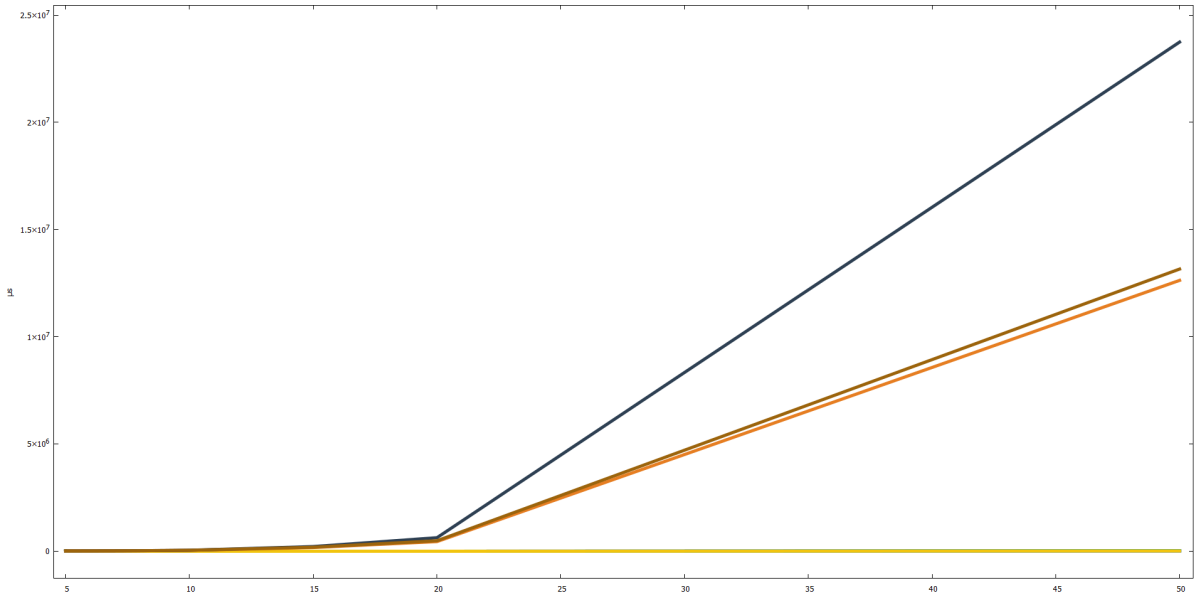


Figure 6.8 – Graphe de comparaison 3 - Temps moyen (en μs)

Dans le graphe 6.8 ci-dessus, nous avons mesuré le temps moyen effectué en fonction de la taille de la grille. Nous remarquons au vu des performances qu'offre la recherche sur plusieurs coups et les résultats, nous pouvons affirmer que la recherche gloutonne avec la deuxième méthode d'évaluation est meilleure que celle-ci. En effet, cette méthode de résolution donne de bons résultats et de bonnes performances.

Nous souhaitons maintenant comparer la recherche gloutonne avec la recherche du plus court chemin. Notons que nous savons que la recherche gloutonne donne de meilleur résultat que la recherche du plus court chemin sur de petite grille (comme nous avons pu le voir dans le graphe 6.6). Ici, nous allons nous intéresser aux résultats qu'offrent ces deux méthodes de résolution sur des grilles de tailles importantes.

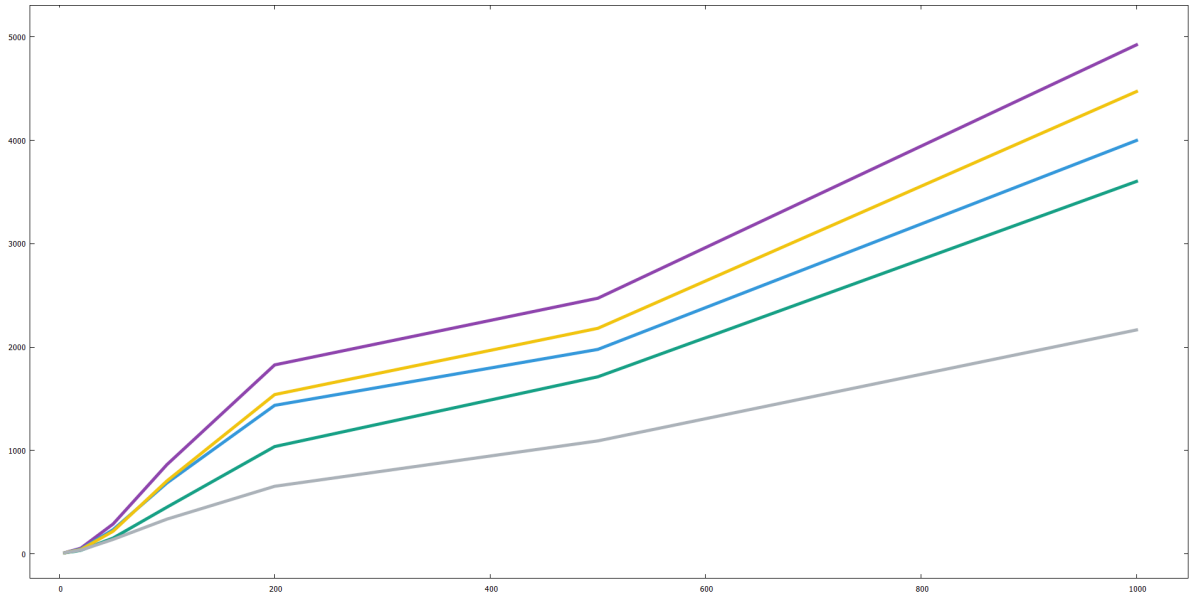


Figure 6.9 – Graphe de comparaison 4 - Nombre de coups moyen

Dans le graphe 6.9 ci-dessus, nous avons mesuré le nombre de coups moyens en fonction de la taille de la grille.

Nous remarquons que la recherche du plus court chemin donne des résultats qui sont bien meilleurs que la recherche gloutonne. De plus, la tendance de la courbe de la recherche du plus court chemin montre que plus on va augmenter la taille de la grille, et plus la recherche gloutonne va s'éloigner de celle-ci et donc de l'optimal. Cependant, nous remarquons que la méthode gloutonne avec la deuxième méthode d'évaluation donne toujours de meilleur résultat que les deux autres.

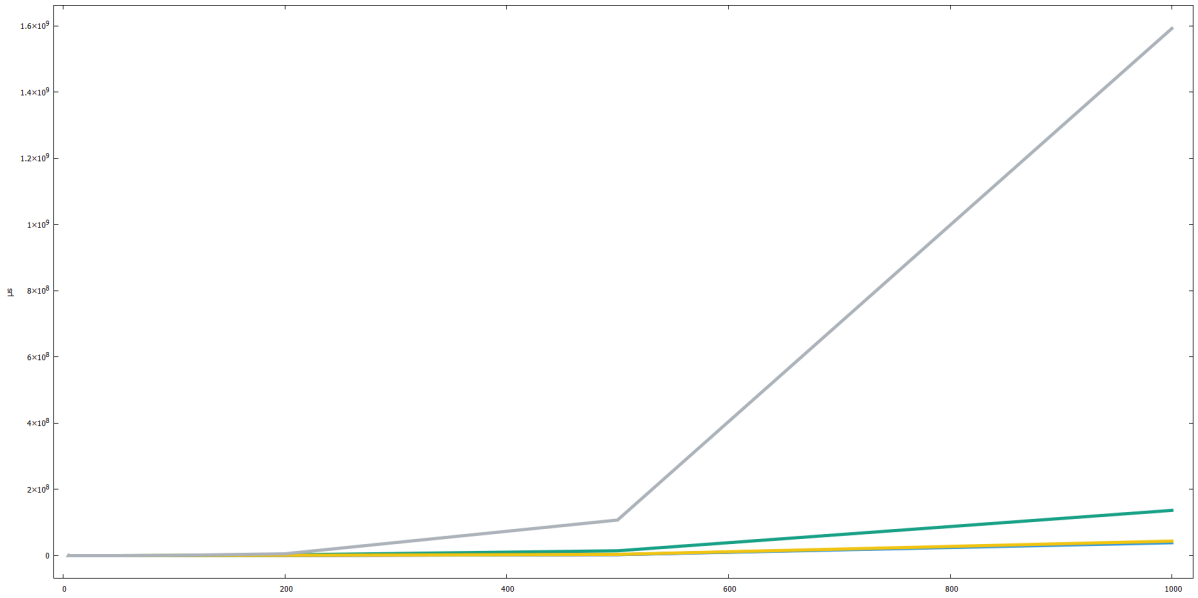


Figure 6.10 – Graphe de comparaison 5 - Temps moyen (en μs)

Dans le graphe 6.10 ci-dessus, nous avons mesuré le temps moyen effectué en fonction de la taille de la grille.

Nous remarquons que la recherche gloutonne a des performances bien meilleures que la recherche du plus court chemin. Cependant, les résultats qu'offre la recherche du plus court chemin par rapport à la recherche gloutonne justifie cette différence.

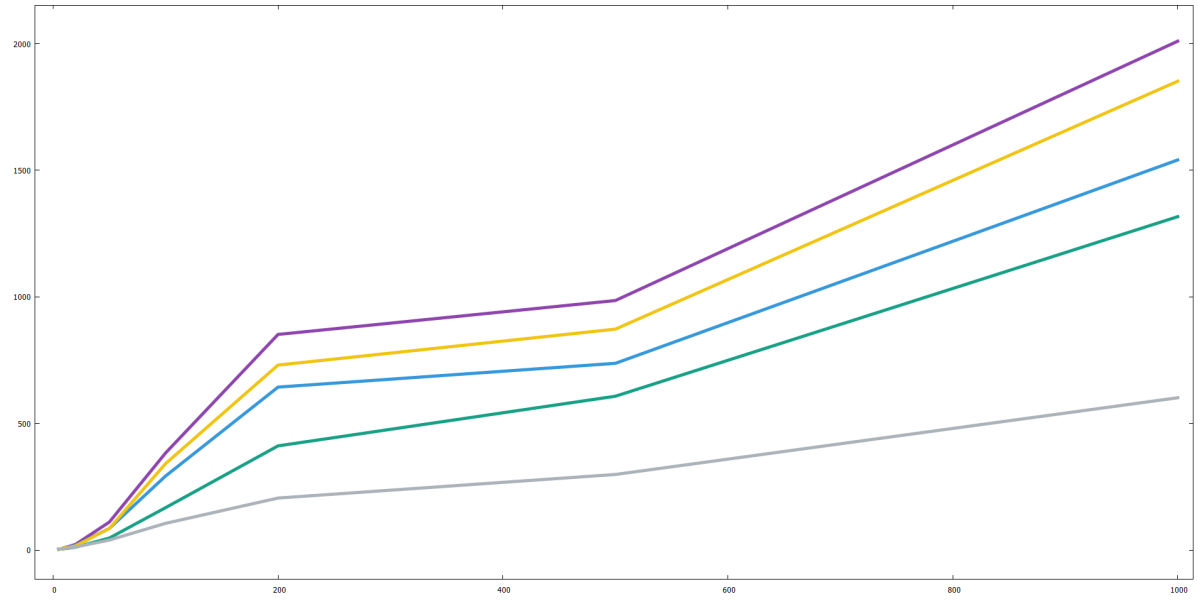


Figure 6.11 – Graphe de comparaison 6 - Écart type du Nombre de coups

Dans le graphe 6.11 ci-dessus, nous avons mesuré l'écart type du nombre de coups effectué

en fonction de la taille de la grille. Cette information est intéressante car elle permet de savoir quelles méthodes de résolutions sont plus sensibles à l'évolution du nombre de couleurs.

Nous remarquons que le nombre de couleur a un impact important sur la recherche gloutonne. De plus, en fonction de la méthode d'évaluation utilisée, le nombre de couleur aura une influence plus ou moins élevé. La recherche du plus court chemin est peu influencé par l'évolution du nombre de couleur. Enfin, nous remarquons que plus la grille est grande, et plus le nombre de couleur a un effet important sur les différentes méthodes de résolution.

Pour conclure cette partie, ce benchmark a pu nous montrer quelles méthodes de résolutions donnaient de bons résultats, lesquelles sont performantes et mauvaises. De plus, nous avons pu mesurer l'impact du nombre de couleurs sur nos différentes méthodes de résolution. Cependant, il serait intéressant de faire un benchmark plus poussé de la recherche exhaustive. En effet, il serait intéressant d'essayer d'avoir la tendance de la courbe optimale et de voir à quel point nos heuristiques sont plus ou moins proche de celle-ci.

Chapitre 7

Conclusion

Après cinq mois de travail, nous pouvons considérer que ce projet a été bénéfique pour nous. Cela nous a donné l'occasion de s'intéresser à un problème NP-difficile. En effet, proposer des méthodes de résolution sur un problème d'optimisation était une première pour nous. De plus, ce projet nous a permis d'apprendre de nouvelles notions en C++ comme les *smart pointeurs*, l'optimisation à la compilation, des éléments de sa librairie standard, etc... Enfin, la découverte du patron de conception "mixin" a été bénéfique pour le groupe. En effet, ce modèle, bien qu'il soit très efficace, son utilisation reste peu répandu et peu utilisé. Cela a donc été très intéressant puisque l'ingénierie logiciel est un point clé de notre cursus.

Durant cette période, nous avons choisi le cycle de vie en spirale comme méthodologie de travail. Cela nous a permis d'ajouter de nouvelles fonctionnalités au fur et à mesure, en évitant le risque d'avoir des erreurs.

Nous avons choisi une bonne méthode de travail accompagnée par une bonne organisation. Cela nous a permis de réaliser toutes les tâches qui étaient présentes dans notre objectif, tel que :

- implémenter Flood-It avec une interface graphique,
- définir un modèle de conception réutilisable,
- implémenter les différentes heuristiques comme : glouton, brute force, ... etc,
- mettre en place un jeu de test et effectuer un benchmark dessus.

Au cours de notre projet, nous avons pu mettre en place un jeu et diverses méthodes pour résoudre celui-ci. Nous avons pu comparer l'efficacité de ces diverses méthodes. Cependant, nous pourrons encore aller plus loin en implémentant l'algorithme tabou (et potentiellement d'autres algorithmes aussi). De plus, puisque nous avons conçu un modèle réutilisable, il serait intéressant de mettre en place un ou deux autres jeux sur lesquels appliquer nos heuristiques.

Bibliographie

[1] Lagoutte, A., Noual, M., & Thierry, E. (2014). Flooding games on graphs. Discrete Applied Mathematics, 164, 532-538.

[2] Métaheuristique. (2018, novembre 28). Wikipédia, l'encyclopédie libre. Page consultée le 08 :17, novembre 28, 2018 à partir de http://fr.wikipedia.org/w/index.php?title=M%C3%A9taheuristique .

[3] Sebastien Le Digabel, (2018), Métaheuristiques. Consulté sur : https://www.gerad.ca/~alainh/Metaheuristiques.pdf.

[4] Jean-Charles Boisson. Modélisation et résolution par métaheuristiques coopératives : de l'atome à la séquence protéique. Recherche opérationnelle [cs.RO]. Université Lille 1, 2008. Français.

[5] Recherche tabou. (2016, avril 2). Wikipédia, l'encyclopédie libre. Page consultée le 18 :31, avril 2, 2016 à partir de http://fr.wikipedia.org/w/index.php?title=Recherche_tabou.

[6] JC Régin - Résolution de Problèmes - L2I - 2011. Consulté sur : http://deptinfo.unice.fr/~regin/cours/cours/ResoPb/ResoPb.pdf .

[7] Hao, J. K., & Solnon, C. (2014). Méta-heuristiques et intelligence artificielle. Chapitre du livre Algorithmes pour l'intelligence artificielle, Editions Cepadues.

[8] Uéverton dos Santos Souza, Fábio Protti, Maise Silva. An algorithmic analysis of Flood-It and Free-Flood-It on graph powers. Discrete Mathematics and Theoretical Computer Science, DMTCS, 2014, Vol. 16 no. 3 (in progress) (3), pp.279–290. hal-01188900.

[9] K. Meeks, A. Scott, The Complexity of Flood-Filling Games on Graphs, Discrete Applied Mathematics 160 (2012) 959–969 .

[10] Templates and Duck Typing. By Andrew Koenig and Barbara E. Moo, June 01, 2005. http://www.drdobbs.com/templates-and-duck-typing/184401971

[11] Garbage Collector (Ramasse-miettes) (Wikipédia) : https://fr.wikipedia.org/wiki/Ramasse-miettes_(informatique)

[12] Fuite de mémoire (Wikipédia) : https://fr.wikipedia.org/wiki/Fuite_de_m%C3%A9moire

[13] Loïc Joly - Pointeurs intelligents - Developpez.com : https://loic-joly.developpez.com/tutoriels/cpp/smart-pointers/

[14] Edsger Dijkstra - Citation : https://dicocitations.lemonde.fr/citation_internaute_ajout/2988.php

[15] Problème du diamant - Wikipédia : https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_diamant

[16] By Ulrich W. Eisenecker, Frank Blinn, and Krzysztof Czarnecki, January 01, 2001 - Mixin-Based Programming in C++ : http://www.drdobbs.com/cpp/mixin-based-programming-in-c/184404445

[17] Mathieu Turcotte - C++ mixins : http://mathieuturcotte.ca/textes/mixins/

[18] Algorithme de Dijkstra — Wikipédia https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra

[19] Présentation des principaux design patterns en C++ - partie clonage - developpez.com : https://come-david.developpez.com/tutoriels/ndps/?page=Clonage

Annexe A

Annexe

A.1 Qt Creator

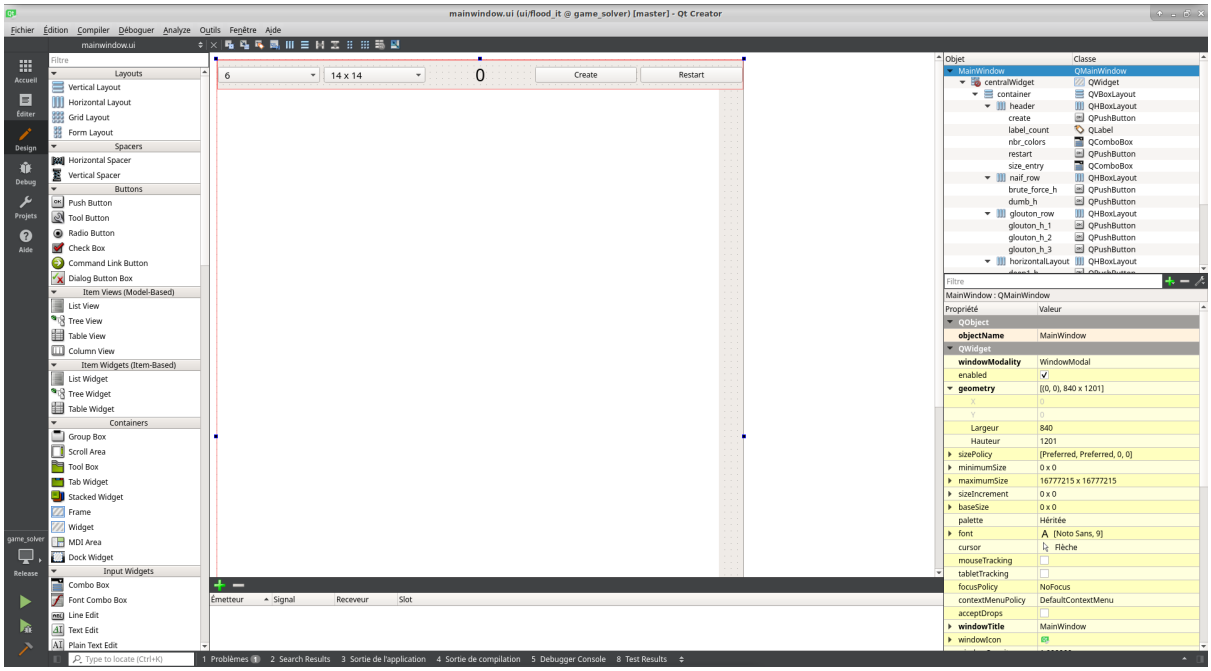


Figure A.1 – Qt Creator : designer

Ceci est l'outil designer proposé par Qt permettant de pouvoir créer des interfaces graphiques simplement.

A.2 Code : benchmark Smart pointer

t3.h :

```
class A
{
protected:
        int a;

public:
        A(const int&);
        virtual int getA() const;
};
```

t3.cpp :

```
#include <iostream>
#include <memory>
#include <vector>
#include <chrono>
#include "t3.h"

using namespace std;
using namespace std::chrono;

A::A(const int& a): a(a)
{
}

int A::getA() const
{
        return a;
}

vector<A*>* array1(const int& count)
{
        vector<A*>* array = new vector<A*>();
        array->reserve(count);

        for (int i = 0; i < count; ++i)
        {
                array->emplace_back(new A(i));
        }

        return array;
}

vector<unique_ptr<A>>* array2(const int& count)
{
        vector<unique_ptr<A>>* array = new vector<unique_ptr<A>>();
        array->reserve(count);

        for (int i = 0; i < count; ++i)
```

```
        {
                array->emplace_back(new A(i));
        }

        return array;
}

vector<shared_ptr<A>>* array3(const int& count)
{
        vector<shared_ptr<A>>* array = new vector<shared_ptr<A>>();
        array->reserve(count);

        for (int i = 0; i < count; ++i)
        {
                array->emplace_back(new A(i));
        }

        return array;
}

auto calc_duration(const auto& t2, const auto& t1)
{
        return (duration_cast<microseconds>(t2 - t1));
}

int main()
{
        int count = 1000000;

        /* ARRAY1 */

        auto start1 = high_resolution_clock::now();
        vector<A*>* o1 = array1(count);
        auto stop1 = high_resolution_clock::now();

        /* ARRAY2 */

        auto start2 = high_resolution_clock::now();
        vector<unique_ptr<A>>* o2 = array2(count);
        auto stop2 = high_resolution_clock::now();

        /* ARRAY3 */

        auto start3 = high_resolution_clock::now();
        vector<shared_ptr<A>>* o3 = array3(count);
        auto stop3 = high_resolution_clock::now();

        /* BENCHMARK */

        cout << "-------------------------------" << endl;
        cout << "Test avec optimisation à la compilation" << endl;
        cout << "-------------------------------" << endl;
```

```
        auto t1 = calc_duration(stop1, start1);
        auto t2 = calc_duration(stop2, start2);
        auto t3 = calc_duration(stop3, start3);

        cout << "create␣raw␣pointer␣:␣" << t1.count() << "us" << endl;
        cout << "create␣\textit{shared␣pointer}␣:␣" << t2.count() << "us"  << endl;
        cout << "create␣shared␣pointer␣:␣" << t3.count() << "us"  << endl;

        cout << "-------------------------------" << endl;

        // Libération de la mémoire manuelle avec des raw pointer

        auto start4 = high_resolution_clock::now();

        for (int i = 0; i < count; ++i)
        {
                delete o1->at(i);
        }

        delete o1;

        auto stop4 = high_resolution_clock::now();

        // Libération automatique de la mémoire avec les smart pointer

        auto start5 = high_resolution_clock::now();
        delete o2;
        auto stop5 = high_resolution_clock::now();

        auto start6 = high_resolution_clock::now();
        delete o3;
        auto stop6 = high_resolution_clock::now();

        auto t4 = calc_duration(stop4, start4);
        auto t5 = calc_duration(stop5, start5);
        auto t6 = calc_duration(stop6, start6);

        cout << "delete␣raw␣pointer␣:␣" << t4.count() << "us"  << endl;
        cout << "delete␣\textit{shared␣pointer}␣:␣" << t5.count() << "us"  << endl;
        cout << "delete␣shared␣pointer␣:␣" << t6.count() << "us"  << endl;

        cout << "-------------------------------" << endl;

        return 0;
}
```

A.3 Cycle de vie en spirale

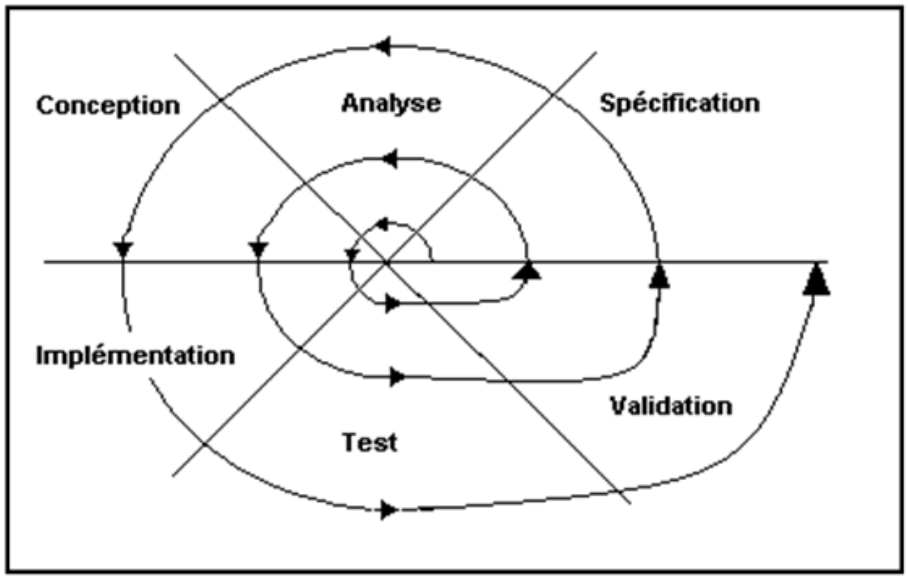


Figure A.2 – Cycle de vie en spirale

Dans ce cycle de vie, une spirale représente l'accomplissement d'une tâche. Lorsqu'une nouvelle tâche doit être effectuée, on rentre dans une nouvelle boucle.

A.4 Résultats - Base de données

Voici le code SQL correspondant à la création de la table nécessaire où sera stocké le résultat issu de notre benchmark :

```
CREATE TABLE data (
  id bigint NOT NULL AUTO_INCREMENT,
  name varchar(20) NOT NULL,
  size int NOT NULL,
  nb_color int NOT NULL,
  nb_move int NOT NULL,
  time int NOT NULL,
  PRIMARY KEY (id)
);
```

Les colonnes correspondent donc à :

- name : nom de la méthode de résolution (+ numéro de la méthode d'évaluation s'il y en a une)
- size : taille de la grille
- nb_color : nombre de couleur composant la grille
- nb_move : nombre de coups effectué par la méthode de résolution
- time : temps effectué par la méthode de résolution (en μs)

A.5 Résultats - calcul des statistiques

Voici le code SQL permettant d'effectuer des statistiques sur le résultat du benchmark :

```
SELECT name, size,
       ROUND(AVG(nb_move)), ROUND(STD(nb_move)),
       ROUND(AVG(time))
FROM data
GROUP BY name, size
ORDER BY name, size ASC
```