Laboratoire de langage C++

Quoridor

R. Absil (abs), J. Beleho (bej), N. Vansteenkiste (nvs), M. Wahid (mwa)

1er Février 2016

HEB - ESI

Résumé

Ce document détaille l'énoncé du projet « Quoridor », un petit jeu au tour par tour basé sur le jeu de société éponyme de Mirko Marchesi. Il détaille la démarche à suivre pour l'implémentation de votre projet, ainsi que les consignes à respecter et les modalités de remise.

Table des matières

1 Introduction **3**

2 Règles du jeu **3**

2.1 Déplacement régulier de pion 4

2.2 Conditions de victoire 4

2.3 Saut de pion 5

2.4 Insertion de mur 6

3 Remises intermédiaires et finale **6**

3.1 Remise console 8

3.2 Remise graphique 9

3.3 Remise d'intelligence artificielle 10

4 Compléments **11**

4.1 Observateur / Observé 11

4.2 Algorithmes d'exploration 12

5 Conclusion **14**

Références **14**

1 Introduction

Dans le cadre du cours `cppl2`, il est demandé de réaliser un projet de grande envergure, par groupes de deux personnes. Les étudiants doivent prendre en charge l'intégralité du projet, de la compréhension de son cahier des charges (le présent document) à son implémentation et ses tests, en passant par son analyse et sa modélisation. Étant donné la taille conséquente du travail demandé, plusieurs remises intermédiaires sont demandées, à des points clés du développement.

L'énoncé du projet de première session de cette année est « Quoridor ». Quoridor est un jeu de plateau créé par Mirko Marchesi et édité par Gigamic [1] se jouant à deux ou quatre joueurs sur un plateau carré. Le but pour chaque joueur est d'atteindre le côté du plateau opposé à son point de départ. Les joueurs ont en outre la possibilité d'insérer des murs sur le plateau de jeu afin de ralentir la progression de leurs adversaires.

Ce document est divisé en plusieurs sections aux objectifs distincts.

— La Section 2 présente en détail les règles du jeu et ce qui est attendu du projet final remis par les étudiants. Plus particulièrement, on y décrit la structure du plateau de jeu, les composants du jeu tels que les pions des joueurs et les murs. Cette section détaille également en profondeur les mécanismes qui les régissent.

— La Section 3 décrit les objectifs intermédiaires à remplir par les étudiants, ainsi que les dates auxquelles des remises partielles doivent être effectuées. Notez que chacune de ces remises est cotée. Cette section décrit également les modalités de remise, caractérisant si oui ou non un projet est « recevable ». Par exemple, un projet remis en retard est non recevable.

— La Section 4 décrit les compléments qui pourraient servir aux étudiants pour la réalisation de ce projet. Plusieurs types de compléments sont fournis : d'ordre technique, comme le design pattern « observateur / observé », et algorithmique, liées à l'exploration de graphe.

— Finalement, la Section 5 conclut ce document et rappelle à l'étudiant les points importants à garder en mémoire.

2 Règles du jeu

Quoridor est un jeu de société créé par Mirko Marchesi et édité par Gigamic. Deux ou quatre joueurs s'affrontent sur un plateau carré de taille 9x9. Les règles officielles du jeu peuvent être trouvées sur le site de l'éditeur [2].

Au début de la partie, un coté est assigné à chaque joueur, et chaque joueur commence avec un pion au milieu de son côté respectif [3]. Classiquement, à deux joueurs, un des joueurs est assigné au côté « nord », et l'autre au côté « sud ». Dans le cas d'un jeu à quatre joueurs, les deux autres côtés sont assignés aux deux autres joueurs.

En début de partie, chaque joueur possède également un nombre déterminé de murs. Habi-

1. Tous droits réservés.
2. http://www.gigamic.com/files/catalog/products/rules/quoridor-classic-fr.pdf - Consulté le 29 Janvier 2016.
3. Ceci implique que le côté d'un plateau a toujours un nombre impair de cases.

tuellement, à deux joueurs (resp. quatre joueurs), chaque joueur possède 10 (resp. 5) murs de longueur 2. Un mur peut être inséré entre deux paires de cases afin d'empêcher la progression de tout joueur entre ces cases.

Notez que dans le cas de l'implémentation qui vous est demandée, la taille du plateau doit être laissée à la discrétion de l'utilisateur. Elle doit cependant respecter les conditions suivantes :
- elle doit être impaire ;
- elle doit être supérieure ou égale à 5,
- elle doit être inférieure ou égale à 19.

Dès lors, le nombre de murs à attribuer à chaque joueur sur un plateau $n \times n$ est le suivant :
1. $n + 1$ murs par joueur en présence de deux joueurs,
2. $\frac{n+1}{2}$ murs par joueur en présence de quatre joueurs.

Notez qu'avec cette formule, à deux joueurs sur un plateau 9×9, chaque joueur possède 10 murs, seulement 5 à quatre joueurs.

Les joueurs jouent à tour de rôle et obligatoirement à chaque tour de jeu. Un joueur peut choisir de déplacer son pion sur le plateau ou d'insérer un mur. Ces déplacements de pions et d'insertion de murs doivent respecter des règles particulières.

2.1 Déplacement régulier de pion

Un pion peut se déplacer latéralement, d'une case à la fois, s'il n'y a pas de murs entre sa case de départ et de destination. Ainsi, les murs doivent être contournés. Un pion ne peut pas sortir du plateau de jeu. Une case ne peut également contenir qu'un unique pion.

Exemple 1. La Figure 1 illustre quelques déplacements autorisés et interdits. On remarque que le pion 1 peut se déplacer sur toutes les cases adjacentes horizontalement ou verticalement. Le pion 2 ne peut pas sortir du plateau et ne peut donc pas de déplacer vers la droite. Le pion 3 ne peut pas se déplacer vers le haut ou la droite, à cause des murs présents sur le plateau, représentés en gras.

2.2 Conditions de victoire

Une partie de Quoridor se termine dès qu'un joueur atteint une des cases du côté opposé à son côté de départ. Un tel joueur est déclaré gagnant. Par exemple, si le joueur 1 commence au milieu du bord nord, il remporte la partie dès qu'il parvient à déplacer son pion sur une des cases du bord sud.

Les joueurs ne peuvent en aucun cas passer leur tour. Un joueur doit donc systématiquement soit déplacer son pion, soit insérer un mur. Dans le cas improbable où un joueur n'aurait plus de murs en sa possession, et ne disposerait d'aucun déplacement valide, la partie est déclarée nulle [4].

4. Notez que ce cas de figure n'est pas précisé dans les règles de base du jeu original.

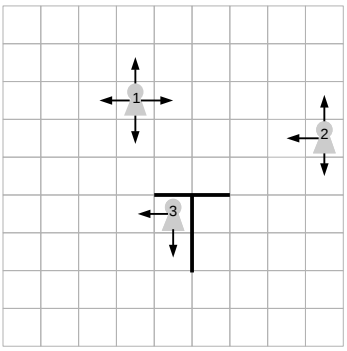

FIGURE 1 – Illustration de déplacements autorisés

2.3 Saut de pion

Dans le cas où un pion se trouve face à un autre pion, au vu des règles décrites ci-dessus, le joueur dont c'est le tour ne peut se déplacer sur la case occupée. Dans ce cas, il peut choisir de faire « un saut de pion », et se placer au delà de son adversaire.

Si la case de destination n'est pas atteignable directement, par exemple si elle est occupée par un pion adverse (dans le cas d'un jeu à quatre joueurs) ou si elle est séparée de la case de départ par un mur, le pion dont c'est le tour peut se déplacer en diagonale dans la direction de son saut. En aucun cas un pion ne peut sauter au dessus de plusieurs autres.

Exemple 2. La Figure 2 illustre plusieurs cas de saut de pions autorisés et interdits. Le pion 2 ne peut pas sauter au dessus du pion 1 à cause du mur supérieur, et ne peut donc que sauter en diagonale vers la gauche en plus de ses déplacements réguliers. Le pion 4 peut sauter au dessus du pion 3. Le pions 5 peut se déplacer en diagonale vers le haut. Quant au pion 6, il peut uniquement se déplacer en diagonale vers le haut, à gauche, en plus de ses déplacements réguliers.

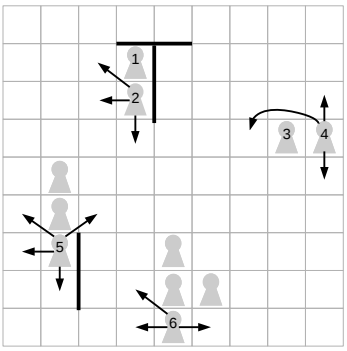

FIGURE 2 – Illustration de sauts de pions autorisés

2.4 Insertion de mur

À la place de déplacer son pion, un joueur peut insérer un mur latéralement entre deux paires de cases. Tous les murs sont donc « de longueur 2 ». Une telle insertion n'est uniquement possible si

1. le joueur insérant le mur possède encore un mur à insérer,
2. le mur est complètement inséré dans le plateau (pas de mur dont l'extrémité dépasse d'un bord du plateau),
3. le mur inséré ne « coupe pas » un autre mur déjà présent sur le plateau,
4. chaque joueur possède toujours un chemin vers une case de son bord de destination après insertion (sans tenir compte des pions adverses).

De plus, les murs ne peuvent être placés sur les bords nord, sud, est ou ouest du plateau.

Exemple 3. La Figure 3 illustre diverses insertions de murs possibles ou interdites. Les murs déjà présents sont indiqués en noir. Les insertions possibles sont indiquées en gris gras, les insertions interdites en pointillé. Les flèches partant des joueurs indiquent dans quelle direction se trouve leur bord de destination.

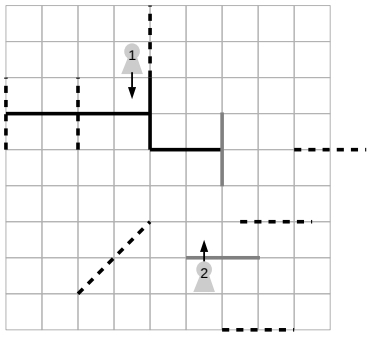

FIGURE 3 – Illustration des insertions de murs possibles

Notez que les situations particulières illustrées à la Figure 4 sont tout à fait possibles (indépendamment du nombre de murs insérés). De tels cas doivent être gérés.

3 Remises intermédiaires et finale

Cette section décrit les remises intermédiaires demandées pour Quoridor. Chacune de ces remises est cotée, aussi, l'étudiant se doit de travailler régulièrement et de remettre ses travaux à temps. Dans la mesure où le projet est effectué par groupe de deux personnes, une remise par groupe est suffisante pour chacune des étapes ci-dessous.

Ces remises permettent de guider les étudiants qui remettraient un travail intermédiaire très insuffisant. Ceci peut arriver lorsque l'étudiant n'a pas fourni la quantité de travail demandée,

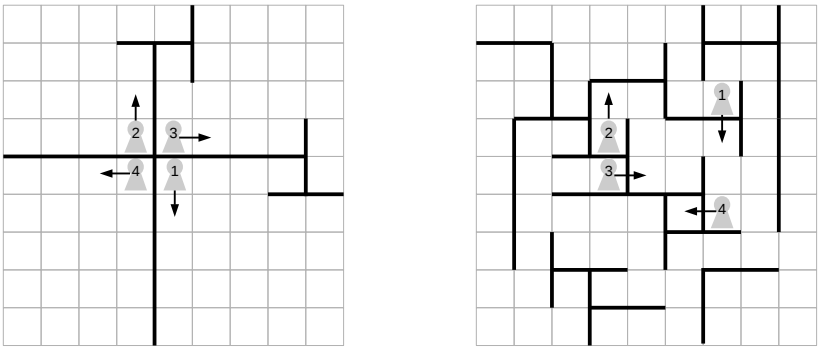

FIGURE 4 – Situations particulières autorisées

ou quand il a fait des choix liés à la modélisation ou l'implémentation de son projet qui sont inadéquats.

Dans ce deuxième cas, les remises intermédiaires offrent la possibilité aux professeurs de corriger l'étudiant afin que les autres remises, en particulier la remise finale, se passent sans encombre.

Plus particulièrement, le travail est divisé en trois étapes, toutes faisant l'objet d'une remise.

1. Une application console complète et fonctionnelle de l'intégralité du jeu. Ceci inclut la documentation du code, ainsi que des tests suffisants.
2. Une application avec interface graphique complète de l'intégralité du jeu.
3. Une application avec interface graphique complète de l'intégralité du jeu incluant une intelligence artificielle pour jouer contre l'ordinateur.

Le contenu de ces remises est détaillé dans les sections suivantes. Notez que chacune de ces « couches » doit être bien distincte : il ne peut y avoir de dépendance de la partie métier vers la partie graphique ou vers l'intelligence artificielle, de l'intelligence artificielle vers l'interface graphique, etc. Ce point vous sera rappelé plusieurs fois dans la description de ces étapes.

Par ailleurs, pour chaque remise, vous devez remettre un rapport expliquant votre travail. Le contenu, la taille et la forme de ce rapport dépendent des consignes que votre maître-assistant vous donnera. Plusieurs référence [6, 7] sont mises à votre disposition pour vous aider dans cette tâche.

Remarquez que quelle que soit l'interface publique des classes produites à chacune des remises, elle ne doit pas permettre à l'utilisateur de laisser le programme dans un état incohérent. Dès lors, si l'exécution d'une fonction requiert qu'une certaine précondition soit respectée (sur le système ou sur les paramètres), vous devez vous assurez que

— soit ladite condition est respectée,
— soit l'exécution de la fonction est stoppée, par exemple via le lancement d'une exception ou l'affichage d'un message d'erreur,
— soit la fonction en question ne modifie pas l'état du système, et l'utilisateur est averti par

exemple via une valeur de retour particulière.

Ainsi, les seules fonctions qui ne testeront pas leurs préconditions associées seront les fonctions à visibilité *privée*.

La finalité de ce projet n'est donc pas uniquement de produire un jeu à interface graphique et intelligence artificielle. C'est également de produire des classes métier que d'*autres* pourraient être amenés à utiliser et à habiller graphiquement selon leurs désirs. Vous devez donc accorder une attention toute particulière à remettre un code qui soit bien documenté, et qui ne soit pas enclin aux erreurs.

Sur un autre sujet, notez que pour toutes les remises, il est inutile de remettre des fichiers compilés (tels que des fichiers objets, des fichiers `.exe`), etc. De plus, le code remis doit être conforme à la norme `C++14`.

De plus, n'indiquez pas de chemin absolu dans la configuration de votre projet (dans le code ou fichiers associés) : le professeur ne possède pas la même arborescence de fichiers que vous.

Enfin, ne remettez pas de code qui soit dépendant du système d'exploitation [5] sur lequel vous développez : votre professeur ne dispose peut-être pas du même système d'exploitation. Notez néanmoins que quel que soit le projet que vous remettez, il doit compiler et pouvoir s'exécuter sans erreurs sur les ordinateurs de l'école.

Finalement, chaque professeur pouvant avoir des exigences particulières quant à ces remises (où remettre, sous quelle forme, etc.), assurez-vous de bien suivre ses consignes !

3.1 Remise console

Une des premières remises à effectuer est une version console du jeu de Quoridor, c'est-à-dire une version complètement fonctionnelle du jeu, dont l'affichage est limité à des impressions dans la sortie standard. En particulier, votre interface doit permettre aux utilisateurs de jouer, ainsi que de choisir leur mode de jeu (à deux ou quatre joueurs, sur un plateau par défaut ou de taille arbitraire).

Notez qu'il est probablement pertinent de procéder par une étape de modélisation dûment réfléchie, au cours de laquelle vous découperez ce projet en classes aux fonctionnalités bien déterminées. Bien que ce ne soit pas explicitement demandé, il est donc inadmissible de remettre un code contenu dans un nombre limité de classes faisant l'intégralité du travail.

Vous êtes libres d'utiliser la modélisation que vous jugez la plus appropriée, mais en restez responsables dans tous les cas. Vous êtes néanmoins forcés d'utiliser le paradigme orienté objet pour votre code, et d'utiliser le design pattern « observateur / observé » pour la gestion des interactions entre les différentes couches logiques de votre programme. Ceci inclut l'utilisation de ce pattern pour la version console. L'étudiant non familier à ce concept trouvera des informations complémentaires en Section 4.1.

5. Par exemple, pas de `#include <windows.h>` dans votre code. Respectez également la casse dans les chemins d'accès aux fichiers.

Notez également qu'en aucun cas une dépendance de la partie « métier » de votre code vers soit son implémentation en console ou graphique ne sera tolérée. Il est donc inadmissible de retrouver des objets de quelque type que ce soit de la librairie `Qt` au sein du cœur de votre programme. Ceci implique également qu'il ne doit pas y avoir de trace du design pattern observateur / observé ou de l'intelligence artificielle au sein du cœur de votre projet. Par ailleurs, les classes métiers n'interagissent pas avec « le monde extérieur » (par définition). Ceci implique l'absence de flux dans les classes métier.

Remarquez également que vous *devez* documenter votre code à cette étape au format `Doxygen`. Vous ne pouvez également pas vous contenter d'une documentation minimale pour vos prototypes, particulièrement si le code qu'ils vont implémenter n'est pas évident. Ceci inclut la documentation des méthodes privées, et les commentaires au sein des algorithmes non triviaux.

C'est également à cette étape que vous devez effectuer les tests (potentiellement unités) de votre code. Ces tests sont loin d'être superficiels : ils apportent de la confiance quant à la justesse de l'implémentation et au bon fonctionnement de votre projet. Similairement, vous ne devez pas hésiter à inclure une liste des bugs que vous avez détectés et que vous n'êtes pas parvenu à résoudre. Une telle honnêteté relève de la maturité, et est préférable à une omission qui peut être incombée soit à de la malhonnêteté, soit à des tests insuffisants.

Cette étape doit être remise lors de la séance de laboratoire de la semaine du 7 mars 2016 au plus tard. Toute remise ultérieure à cette date ne sera pas reçue et sera donc jugée comme nulle.

3.2 Remise graphique

La deuxième remise à effectuer concerne une remise graphique de votre projet, qui doit être basée sur les classes métier de votre première remise. Ceci signifie qu'une interface utilisateur utilisant la librairie `Qt` doit être implémentée afin de pouvoir jouer à Quoridor selon les règles décrites dans ce projet.

De la même manière qu'en console, votre interface graphique doit permettre à l'utilisateur de choisir son mode de jeu (par défaut à deux joueurs, par défaut à quatre joueurs, ou sur un plateau personnalisé).

Encore une fois, les étudiants doivent utiliser le design pattern « observateur / observé » pour la gestion des interactions entre les différentes couches logiques de leur programme. Vous devez donc vous en servir à la fois pour les événements graphiques au sein de votre interface, tels que « un utilisateur a pressé un bouton », que ceux qui ont été utilisés dans la remise console.

Similairement à la remise console, vous ne devez pas avoir de dépendance de la partie métier vers la partie graphique ou même la couche « observateur / observé ». Idéalement, vous n'aurez pas besoin de modifier une seule ligne des classes métier précédemment remises pour concevoir votre interface graphique.

La prise en main et jouabilité de votre application est laissée à votre appréciation, vous êtes donc libre de choisir le type d'interaction avec l'interface. Ceci peut donc être au clavier, à la souris, ou les deux.

De la même manière, vous pouvez vous contenter d'une version « minimale » de votre interface, ou implémenter des fonctionnalités additionnelles, telles que

— un aperçu des cases disponibles pour un déplacement de pion ;
— une mise en évidence des insertions invalides pour un mur, plutôt qu'un message d'erreur « brutal » ;
— une suggestion d'un coup à un joueur, une fois l'intelligence artificielle implémentée ;
— une possibilité de sauvegarde de chargement d'une partie en cours ;
— une possibilité de jouer sur un plateau non carré, etc.

Dans tous les cas, n'implémentez pas de fonctionnalités additionnelles de votre projet *avant* d'avoir implémenté *la totalité* de ce qui est exigé par ce document et votre maître-assistant.

Par ailleurs, outre la liste des bugs éventuels, indiquez dans votre rapport toutes les modifications des classes métier que vous avez réalisées ainsi que les raisons de ces changements.

Cette étape doit être remise lors de la séance de laboratoire de la semaine du 11 avril 2016 au plus tard. Toute remise ultérieure à cette date ne sera pas reçue et sera donc jugée comme nulle.

3.3 Remise d'intelligence artificielle

La dernière remise de votre projet, qui fait également office de remise finale, consiste en l'implémentation d'au moins une intelligence artificielle pour votre jeu.

Le but de ce composant est de permettre simplement à un ou plusieurs joueurs de jouer contre l'ordinateur, ou de permettre à deux (ou quatre) intelligences artificielles de s'affronter. Dans ce cas de figure, il peut être utile, au sein de votre interface, de permettre de choisir le mode de jeu, et en mode « IA contre IA [6] », de choisir les personnalités des intelligences artificielles, si vous en avez codé plusieurs types.

Encore une fois, notez que si vous avez dûment réfléchi à la modélisation de votre code, vous n'aurez pas besoin de modifier une seule ligne de la partie métier de votre code pour implémenter l'intelligence artificielle. Vous devrez toutefois peut-être ajouter quelques composants graphiques pour offrir les choix mentionnés ci-dessus à l'utilisateur.

Vous êtes libres d'implémenter le type d'intelligence artificielle que vous souhaitez, d'une énumération partielle des possibilités à une technique personnalisée, aux techniques plus complexes telles que les algorithmes MinMax et l'élagage Alpha Beta. Notez néanmoins qu'une IA « non naïve » sera mieux notée.

Quelle que soient vos implémentations, vos IA possèdent soit

— un maximum de 5 secondes CPU pour jouer, par coup,
— globalement 15 minutes de temps de jeu par partie. Ce temps est également attribué aux joueurs humain. Quand c'est le tour du joueur i de jouer, son temps restant diminue. Une fois qu'il a joué, son temps restant s'arrête. Si un joueur n'a plus de temps, il a automa-

6. Dans ce cas, il est recommandé de cadencer l'exécution de votre programme (via `std::this_thread.sleep_for`, par exemple) afin de ne pas voir uniquement le résultat final.

tiquement perdu.

Sans l'une de ces contraintes, on pourrait simplement énumérer toutes les possibilités de coup (ce qui prend un temps considérable) et jouer « parfaitement » en choisissant les coups qui conduisent à la victoire.

Dans tous les cas, vous devez fournir des tests pour votre intelligence artificielle, ne fusse pour s'assurer qu'elle joue un coup (et un seul), qu'elle ne réfléchit pas indéfiniment, etc.

Au cas où vous implémentez plusieurs IA, vous pouvez également fournir des résultats statistiques liés à leur utilisation l'une contre l'autre, afin de déterminer laquelle semble la plus performante.

Cette étape doit être remise pour le 10 mai 2016 au plus tard, lors de la séance de révisions. Toute remise ultérieure à cette date ne sera pas reçue et sera donc jugée comme nulle.

4 Compléments

Cette section détaille à l'étudiant intéressé des compléments d'ordre technique et algorithmique qui pourraient lui être utile dans l'élaboration de son projet. En premier lieu, elle détaille le design pattern « Observateur / Observé », et offre ensuite une courte introduction aux algorithmes d'exploration de graphe.

4.1 Observateur / Observé

Cette section détaille brièvement à l'étudiant intéressé le design pattern « Observateur / Observé » qu'il est demandé d'utiliser tout au long de l'élaboration de ce projet.

Le design pattern « Observateur / Observé » est une pratique courante de découpe simple du code au sein d'une application. Dans une telle application, l'environnement modélisé peut subir des modifications, et ces modifications doivent entraîner des actions en conséquence, par exemple une mise à jour de l'interface affichant l'environnement.

Dès lors, on définit deux composants : le *sujet* ou l'*observable*, en l'occurrence ici l'environnement, qui maintient une liste d'*observateurs*, ici les composants qui vont se charger de monitorer l'environnement et d'effectuer les actions appropriées à chaque modification. En général, les observateurs possèdent des méthodes associées à chaque type de modification.

Notez que dans une implémentation « propre » d'un programme utilisant ce design pattern, l'environnement ne dépend pas du sujet, ils sont découplés. Le sujet est simplement un composant qui soit hérite de l'environnement, soit utilise une relation de composition. Par ailleurs, chaque observateur rattaché au sujet doit être manuellement ajouté avant de commencer à monitorer l'environnement, et supprimé quand ce monitoring est terminé.

Cette technique est très utilisée, notamment dans les librairies d'interface utilisateur comme `Swing` en `Java` et `Qt` en `C++`. Par exemple, en `Java`, un `JButton` est observé, on monitore le fait

que ce bouton soit pressé ou non. L'observateur en question est l'`ActionListener` associé, qui va effectuer un traitement (`actionPerformed`) à chaque fois que le bouton est pressé.

Votre professeur présentera pendant les séances de laboratoire ce design-pattern, sur base d'exemples qu'il mettra à votre disposition. L'étudiant souhaitant approfondir ce qui aura été exposé est libre de consulter la littérature [3–5] disponible à ce sujet.

4.2 Algorithmes d'exploration

L'exploration d'un espace est une technique très utilisée dans de nombreuses applications informatiques, que ce soit pour le routage de paquets sur un réseau, l'optimisation d'une fonction mathématique, etc. Diverses techniques existent pour ce problème, aussi variées en caractéristiques qu'en complexité.

Dans le cadre de Quoridor, on s'intéresse aux algorithmes liés à l'exploration du modèle *graphe*, que vous avez vu dans votre cours de mathématique. La définition suivante rappelle ce concept.

Définition 1. Un graphe non-orienté G est un couple (V, E) où
— V est un ensemble fini non vide d'éléments appelés *sommets*,
— $E \subseteq V \times V$ est un ensemble de paires non ordonnées de sommets appelées *arêtes*.
Si (u, v) est une arête de G, on dit que v est un *voisin* de u. L'ensemble des voisins d'un sommet est appelé le *voisinage* de ce sommet.

L'utilisation du modèle de graphe dans le cadre de Quoridor est adaptée, dans la mesure où le plateau de jeu peut être modélisé comme tel. À certains endroits, au vu de ce qui est demandé dans l'implémentation, vous aurez besoin d'explorer le plateau de jeu, c'est-à-dire de faire visiter chacune des cases à un algorithme pour effectuer un traitement approprié.

La suite de cette section décrit comment explorer un graphe. L'étudiant intéressé est libre de s'inspirer de cette technique pour son projet, voire de compléter ses connaissance à ce sujet en consultant les livres d'Aho *et al.* [1] ou de Cormen *et al.* [2].

Similairement à ce qui est décrit ci-dessus, le but est donc de visiter chacun des sommets du graphe une unique fois en suivant les arêtes. À chaque sommet rencontré, un traitement peut être effectué. Ce traitement peut simplement consister à « marquer » les sommets visités pour ne pas revenir sur un sommet déjà visité, compter, maintenir une propriété sur une structure de donnée, etc.

Néanmoins, ce traitement dépend souvent de l'ordre dans lequel les sommets sont visités. Pour cette raison, il existe deux stratégies principales d'exploration dans la littérature :
— *en largeur d'abord* (BFS) : l'idée de base est d'explorer complètement le voisinage d'un sommet avant de poursuivre l'exploration.
— *en profondeur d'abord* (DFS) : intuitivement, l'exploration se poursuit aussi loin que possible avant de « revenir en arrière ». Ce type d'exploration est un exemple particulier d'*algorithme en backtracking*.

Exploration en largeur

L'exploration en largeur d'un graphe est souvent mise en œuvre itérativement, à l'aide d'une *file*. Dans cette structure de donnée, l'ajout d'un élément se fait « à la fin » de la file, et la suppression d'un élément « au début ». Ainsi, un élément ajouté avant un autre dans une file sera également supprimé avant. C'est le même mécanisme que les files de clients aux caisses d'un supermarché : les premiers arrivés en caisse sont les premier servis.

L'idée de base est d'ajouter des sommets non marqués dans la file, et de supprimer un à un les éléments de la file, en y ajoutant à chaque fois les voisins non marqués des sommets supprimés. Ainsi, on explore itérativement tous les sommets du graphe. Ce principe est formellement illustré en pseudo-code à l'Algorithme 1.

Algorithme 1 Exploration en largeur d'abord

Entrée(s) : $G = (V, E)$, un graphe non orienté.
Sortie(s) : $\varnothing$, les sommets du graphe ont été explorés.

```
1: Q ⟵ file vide
2: pour chaque sommet v non marqué de G faire
3:     marquer v
4:     enfiler v dans Q
5:     BFS-Search()

6: procédure BFS-Search()
7: tant que Q n'est pas vide faire
8:     Défiler v de Q
9:     pour chaque sommet u non marqué tel que (v, u) ∈ E faire
10:        marquer u
11:        enfiler u dans Q
```

Évidemment, dans cet algorithme, à chaque fois qu'un sommet non marqué est visité, un traitement relatif à ce sommet peut être effectué.

Exploration en profondeur

L'exploration en profondeur d'un graphe, *a contrario* d'en largeur, est souvent mise en œuvre récursivement, sans l'utilisation d'une structure de donnée annexe. Ce mécanisme peut néanmoins être mis en œuvre itérativement à l'aide d'une *pile*.

Intuitivement, à chaque fois que l'on rencontre un sommet non-marqué, on relance un appel récursif d'exploration, et l'on visite ainsi tous les sommets du graphe. Ce principe est formellement illustré en pseudo-code à l'Algorithme 2.

Encore une fois, dans cet algorithme, à chaque fois qu'un sommet non marqué est visité, un traitement relatif à ce sommet peut être effectué.

Algorithme 2 Depth First Search

Entrée(s) : $G = (V, E)$, un graphe non orienté.
Sortie(s) : $\varnothing$, les sommets du graphe ont été explorés.

1: **pour chaque** sommet v non marqué de G **faire**
2: marquer v
3: DFS-Search(v)
4: **procédure** DFS-Search(v)
5: **pour chaque** sommet u non marqué tel que $(v, u) \in E$ **faire**
6: marquer u
7: DFS-Search(u)

5 Conclusion

L'énoncé du projet a été décrit dans son intégralité, en détaillant chacun des composants qui le constituent, ainsi que les exigences liées aux fonctionnalités demandées. Pour rappel, ce projet est effectué par groupe de deux étudiants, et est soumis à plusieurs remises intermédiaires à respecter, la première étant durant la semaine du 7 mars 2016, la deuxième durant la semaine du 11 avril 2016, et la dernière étant le 10 mai 2016 exactement.

Le projet remis doit être découpé en couches indépendantes, doit utiliser le design pattern observateur / observé, ne doit pas être enclin aux erreurs et doit être documenté de manière suffisante.

De plus, chaque maître-assistant peut avoir des exigences particulières, les étudiants doivent donc se renseigner par rapport à ces particularités.

Références

[1] A. Aho, J. Ullman, and X. Cazin. *Concepts fondamentaux de l'informatique*. Sciences sup. Dunod, 1996.

[2] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[3] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002. https://drive.google.com/file/d/0B7-vsG7s7TCIMlBzN0dJOHVtMlE/edit - Consulté le 12 Janvier 2016.

[4] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head First Design Patterns*. O'Reilly, 2004. http://www.sws.bfh.ch/~amrhein/ADP/HeadFirstDesignPatterns.pdf.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. http://www.uml.org.cn/c%2B%2B/pdf/DesignPatterns.pdf - Consulté le 12 Janvier 2016.

[6] T. Hengl and M. Gould. The unofficial guide for authors, 2006. http://tinyurl.com/hanfxbp - Consulté le 12 Janvier 2016.

[7] H. Mélot. Élments de rédaction scientifique en informatique. http://informatique.umons.ac.be/algo/redacSci.pdf - Consulté le 12 Janvier 2016.