

MIASHS_IC 2023-2024

PROJET IA QUORIDOR

Di Malta Gaël ; FAUCHON Louis-Bastien ; RASOAMANANA Rasambaharinosy Nathanaël

13 Décembre 2023

I. Table des matières

Présentation du jeu	3
Objectif :	3
Entrée de jeu :	3
Déroulement :	3
Le déplacement d'un pion :.....	3
Le placement d'un mur :.....	3
Notre organisation.....	4
La découverte du jeu (semaine du 16 au 22 Octobre 2023)	4
La définition des principaux concepts à intégrer (semaine du 6 au 12 Novembre 2023).....	4
La modélisation du fonctionnement du jeu (semaine 13 au 26 Novembre 2023).....	4
Le codage du jeu (du 27 Novembre au 16 Décembre 2023)	4
Algorithmes	5
Les algorithmes de fonctionnement du jeu.....	5
Les algorithmes pour le traitements de l'intelligence artificielle	8
Les difficultés rencontrées.....	10
Penser le mode de fonctionnement du jeu	10
Implémentation d'une recherche heuristique	10
Toujours permettre au pion d'avoir une issue vers une des cases de la ligne à atteindre.	11
Implémentation d'une IA finalement basée le BFS et recherche du chemin optimisé pour le joueur.	
.....	11
Conclusions.....	11
Un guide rapide d'utilisation de l'interface	12
Annexe	14
Historique d'organisation	14
Idées d'interfaces du jeu	14
Notre idée de solution pour A*	14
Aides pour l'algorithme A*	14
Modélisations d'une idée d'algorithme pour éviter l'isolation d'un joueur :.....	15

II. Présentation du jeu

Nous avions pour idée la reproduction du jeu Quoridor. Il s'agit d'un jeu de stratégie qui se joue sur un plateau carré avec des cases. Deux joueurs doivent jouer tour à tour avec chacun un pion et un certain nombre de murs.

Le jeu se déroule ainsi :

1. Objectif :

Chaque joueur tente de déplacer son pion pour arriver dans une des cases de la ligne de départ du joueur adverse. Cependant, il doit aussi empêcher le pion de son adversaire d'arriver dans une des cases de sa ligne de départ et doit poser des murs pour cela.

2. Entrée de jeu :

D'entrée de jeu, le pion de chaque joueur est placé à l'opposé de l'autre. L'un se trouvant sur une case de la première ligne du plateau de jeu et l'autre sur la case opposée, dans la dernière ligne du plateau de jeu.

3. Déroulement :

A chaque tour, le joueur a le choix entre deux actions, soit de déplacer son pion, soit de poser un mur. Il n'a donc droit qu'à une seule action par tour et dès qu'une action est faite, c'est à l'autre joueur de jouer.

4. Le déplacement d'un pion :

Le pion peut être déplacé d'une seule case à une autre et le déplacement vers une case se trouvant à la diagonale n'est pas autorisé. Tant que c'est possible, le pion peut donc être déplacé :

1. Horizontalement : vers la case adjacente à sa gauche ou vers la case adjacente à sa droite
2. Verticalement : vers la case adjacente du haut ou vers la case adjacente du bas

Lorsque le pion adverse se trouve dans une case dans laquelle le pion peut être déplacé, le joueur peut choisir de passer par-dessus le pion adverse pour mettre son pion dans la case suivante.

Le pion ne peut pas passer par-dessus un mur. C'est-à-dire que si un mur est placé entre la case dans laquelle se trouve le pion et une case adjacente où le déplacement est autorisé alors le déplacement vers cette case est interdit.

5. Le placement d'un mur :

Un mur peut-être placé horizontalement ou verticalement. Cependant, il doit toujours être possible pour le pion adverse d'avoir une issue vers une des cases de la ligne de départ du joueur adverse. Le pion ne devrait donc jamais se retrouver clôturé dans un ensemble de murs (ou un ensemble de murs et les bordures du plateau de jeu).

Quand le joueur n'a plus de mur à sa disposition, le déplacement du pion devient sa seule action possible lorsque c'est son tour de jouer.

III. Notre organisation

Notre organisation peut se résumer en quatre étapes

1. La découverte du jeu (semaine du 16 au 22 Octobre 2023)

Durant cette semaine nous avons cherché à comprendre le jeu et à en assimiler les règles et les objectifs.

Les premières idées pour l'interface et les technologies à utiliser sont venues et nous en avons fait un brainstorming dans un premier temps.

2. La définition des principaux concepts à intégrer (semaine du 6 au 12 Novembre 2023)

Pendant cette semaine, nous avons pensé chacun de notre côté la conception du jeu et nous avons pu en définir les classes dont aurait besoin :

- _ une classe Grille qui représente le plateau du jeu
- _ une classe Joueur qui contient les caractéristiques du joueur (les coordonnées du pion, le nombre de murs en possession, les actions,...)
- _ une classe Mur qui contient les caractéristiques des murs
- _ une classe Jeu qui contient la boucle de jeu

A ce stade, nous n'avons pas encore commencé à coder.

3. La modélisation du fonctionnement du jeu (semaine 13 au 26 Novembre 2023)

Nous avons défini un mode de fonctionnement du jeu inspiré de la bataille navale. L'idée étant de pouvoir produire un jeu qui fonctionne et qui soit cohérent dans le temps qui nous restait.

4. Le codage du jeu (du 27 Novembre au 16 Décembre 2023)

Nous nous sommes mis à coder en fonction des idées que nous avons mis en place en amont. Pendant le codage, certains aspects de base qu'on a pensé ont dû être modifiés, les caractéristiques et les méthodes dans les différentes classes ont été ajustées, la confrontation à certaines difficultés concernant la conception des fonctionnements du jeu est apparue,...

Mais en fin de compte nous avons un jeu qui fonctionne avec une interface graphique et surtout différents modes de jeu dont notamment la possibilité de jouer avec un joueur artificiel.

IV. Algorithmes

Nous avons défini 8 classes :

1. **Une classe Grille**, Cette classe représente toutes caractéristiques de la grille pendant la partie ainsi que les fonctions clé du jeu. Elle gère l'affichage de la grille lors d'une partie, vérifie la cohérence des actions faites par les joueurs et fait tourner toutes les fonctions nécessaires au bon fonctionnement du jeu.
2. **Une classe Joueur**, contenant les coordonnées du pion, le nombre de murs en possession et le nom du joueur.
3. **Une classe Mur**, contenant les attributs nécessaires à celui-ci pour pouvoir être identifié ou placé sur la grille.
4. **Une classe IA**, qui contient l'ensemble des méthodes de calcul automatique de distances, et de chemins optimal. Cette classe est aussi un type de joueur permettant de coder de nouveaux comportements.
5. **Une classe IA2**, dans laquelle se trouve notre algorithme de simulation d'actions automatiques pour que le joueur puisse jouer avec une intelligence artificielle.
6. **Une classe Jeu** qui contient la boucle de jeu et importe tous les autres éléments.
7. **Une classe Node** qui représente les nœuds pour tous les traitements de l'IA. Elle est surtout nécessaire pour le fonctionnement de l'algorithme du BFS et pour la grille.
8. **Une classe Graphique** pour notre interface de jeu graphique.

Voici quelques algorithmes du jeu : *main, les fonctions clé (mouvement du pion et placement des murs) et l'IA*.

1. Les algorithmes de fonctionnement du jeu

Algorithme du fonctionnement principal du jeu : main, classe Jeu

```
DEBUT
partieEnCours = true
Tant que (partieEnCours = true) //BOUCLE DE LA PARTIE EN COURS
|   Afficher joueur courant
|   actionError = true
|   Tant que (actionError = true) //BOUCLE DU CHOIX D'ACTION (bouger le pion, placer un mur, quitter la partie)
|   |   Saisir Action
|   |   Si (Action = « bouger »)
|   |       moveError = true
|   |       Tant que (moveError = true) //BOUCLE DU CHOIX DU MOUVEMENT DU PION (haut, droite,...)
|   |       |   Saisir Direction
|   |       |   Si (Direction = « haut »)
|   |           |       Si (mouvement est possible)
|   |               |           Appel fonction movePlayer()
|   |               |           Message « mouvement effectué »
|   |               |           moveError = false //SORTIR DE CHOIX MOUVEMENT
|   |               |           actionError = false //SROTIR DE CHOIX ACTION
|   |       Sinon
|   |           |       Message « Impossible de bouger vers le haut »
|   |       Fin SI
```

```

    |           |
    |           | Sinon
    |           |   Si (Direction = « droite »)
    |           |       Si (mouvement est possible)
    |           |           Appel fonction movePlayer()
    |           |           Message « mouvement effectué »
    |           |           moveError = false
    |           |           actionError = false
    |           |       Sinon
    |           |           Message « Impossible de bouger vers le haut »
    |           |       Fin SI
    |           Sinon
    |           Si (Direction = « bas »)
    |               Si (mouvement est possible)
    |                   Appel fonction movePlayer()
    |                   Message « mouvement effectué »
    |                   moveError = false
    |                   actionError = false
    |               Sinon
    |                   Message « Impossible de bouger vers le bas »
    |               Fin SI
    |           Sinon
    |           Si (Direction = « gauche »)
    |               Si (mouvement est possible)
    |                   Appel fonction movePlayer()
    |                   Message « mouvement effectué »
    |                   moveError = false
    |                   actionError = false
    |               Sinon
    |                   Message « Impossible de bouger vers le gauche »
    |               Fin SI
    |           Sinon
    |           Si (Direction = « retour »)
    |               moveError = false
    |           Sinon
    |               Message « Mot non reconnu, veuillez réessayer »
    |           Fin Si
    |       Fin Tant que
    |   Sinon
    |       Si (Action = « quit »)
    |           moveError = false
    |           actionError = false
    |       Fin Si
    |   Fin Si
    |   Si (Action = « mur »)
    |       wallError = true
    |       Tant que (wallError = true) // BOUCLE DU CHOIX DE L'EMPLACEMENT DU MUR (vert, horiz)
    |           Appel fonction getEmplacementMur() //Récupérer l'emplacement du mur
    |           Saisir centreMur //coordonnées du centre du mur
    |           Si (centreMur = « retour »)
    |               wallError = false //SROTIR DE CHOIX EMPLACEMENT
    |           Sinon
    |               Pour (s parcourant Emplacement)
    |                   Si (s = centreMur)
    |                       directionError = true
    |                       Tant que (directionError = true)
    |                           Saisir Direction
    |                           Si (Direction = « v » ou Direction = « h »)
    |                               Si (mur est possible)
    |                                   Appel fonction placerMur()
    |                                   directionError = false
    |                                   wallError = false
    |                                   actionError = false
    |               Fin Si

```

```

| | | | | | | | Sinon
| | | | | | | | Message « Erreur lors de
| | | | | | | | l'emplacement du mur, choisir un
| | | | | | | | autre emplacement »
| | | | | | | directionError = false
| | | | | | Fin Si
| | | | | Fin Tant que
| | | | Fin Si
| | | | Fin Pour
| | | Fin Si
| | Fin Tant que
| Sinon
| | Demander Action
| Fin Si
Fin Tant que
Afficher la grille
appel fonction switchPlayer() // Changer de joueur
Si (coordonnéeY J1 = taille de la grille – 1)
| Message « J1 gagne la partie »
| partieEnCours = false //SROTIR DE PARTIE EN COURS
Sinon
Si (coordonnéeY J2 = 0)
| Message « J2 gagne la partie »
| partieEnCours = false
Fin Si
Fin Tant que
FIN

```

Algorithme pour le déplacement du pion

```

DEBUT
| Remplacer le contenu de la case de la grille contenant le nom du joueur courant par un blanc « »
| Si (direction == 0) // paramètre correspondant à «haut»
| | On enlève 2 à Y du joueur courant
| | On met le nom du joueur courant sur la case de la grille
| | ayant comme coordonnées X et Y du joueur courant
| Sinon
| | Si (direction == 1) // paramètre correspondant à «droite»
| | | On ajoute 2 à X du joueur courant
| | | On met le nom du joueur courant sur la case de la grille
| | | ayant comme coordonnées X et Y du joueur courant
| | Sinon
| | | Si (direction == 2) // paramètre correspondant à «bas»
| | | | On ajoute 2 à Y du joueur courant
| | | | On met le nom du joueur courant sur la case de la grille
| | | | ayant comme coordonnées X et Y du joueur courant
| | Sinon
| | | Si (direction == 3) // paramètre correspondant à «gauche»
| | | | On enlève 2 à X du joueur courant
| | | | On met le nom du joueur courant sur la case de la grille
| | | | ayant comme coordonnées X et Y du joueur courant
| Fin Si
FIN

```

Algorithme pour les placements des murs

DEBUT

```
| On récupère les coordonnées du centre du mur
| On enlève un mur au joueur courant
| Si (direction = « v »)
|   | mettre « X » dans les coordonnées(X, Y) du centre du mur
|   | mettre « X » dans les coordonnées(X, Y + 1) du centre du mur
|   | mettre « X » dans les coordonnées(X, Y – 1) du centre du mur
| Sinon
|   Si (direction = « h »)
|     | mettre « X » dans les coordonnées(X, Y) du centre du mur
|     | mettre « X » dans les coordonnées(X + 1, Y) du centre du mur
|     | mettre « X » dans les coordonnées(X – 1, Y) du centre du mur
| Fin Si
| Afficher le nombre de mur du joueur courant
```

FIN

2. Les algorithmes pour le traitements de l'intelligence artificielle

Exploration de toutes les cases accessibles depuis un joueur

DEBUT

```
| Création d'une ArrayList de String "casePere" pour stocker les pères des cases que nous allons explorer (les
cases |   depuis laquelle la case en question a été découverte)
|   On ajoute la première cases depuis laquelle la méthode a été appelée dans casePere pour ne pas avoir un
élément |     vide
|   For( i plus petit que la taille de l'ArrayList casesVu )
|     |   On crée une String caseCourante qu'on instancie avec la valeur de casesVu à l'indice i
|     |   On créé 4 String : caseGauche,casesHaut,casesDroite et caseBas que l'on calcule à l'aide d'autres petites
|     |   méthodes de la classe ) partir de la caseCourante et de la grille de jeu
|     |   Si (caseGauche différent de "")// On vérifie que la case voisine est dans la grille
|     |     |   Si( la case n'est pas déjà dans casesVu)
|     |     |     |   on add à casesVu caseGauche
|     |     |     |   on add la caseCourante dans casesPere
|     |   Si (caseHaut différent de "")// On vérifie que la case voisine est dans la grille
|     |     |   Si( la case n'est pas déjà dans casesVu)
|     |     |     |   on add à casesVu caseDroite
|     |     |     |   on add la caseCourante dans casesPere
|     |   Si (caseDroite différent de "")// On vérifie que la case voisine est dans la grille
|     |     |   Si( la case n'est pas déjà dans casesVu)
|     |     |     |   on add à casesVu caseDroite
|     |     |     |   on add la caseCourante dans casesPere
|     |   Si (caseBas différent de "")// On vérifie que la case voisine est dans la grille
|     |     |   Si( la case n'est pas déjà dans casesVu)
|     |     |     |   on add à casesVu caseBas
|     |     |     |   on add la caseCourante dans casesPere
|   On crée un tableau d'ArrayList de String résultat // de taille 2
|   on stocke l'arrayList des casesVu dans résultat[0]
|   on stocke les casesPere dans résultat[1]
|   return résultat
```

FIN

Vérification que la pose d'un mur n'isole pas de cases de la grille

DEBUT

```
| On crée 2 faux joueurs placés sur la grille
| On crée une copie de la grille de jeu
| FOR (chaque ligne de la vraie grille)
|   | FOR (chaque colonne de la vraie grille)
|   |   | On copie la case de la vraie grille dans la fausse
|   | On place le mur demandé par le joueur qui a appelé cette méthode dans la fausse grille
|   | On crée une variable int qui contient le nombre de cases visitable par les joueurs dans la grille
|   | "nbCasesAVisiter"
|   | On récupère la case de l'emplacement actuel du joueur qui a appelé la méthode et on la stocke dans une variable
|   | caseDepart
|   | On crée une ArrayList de string qui contiendra le parcours des cases "caseVu"
|   | On ajoute caseDepart dans casesVu
|   | On appelle la méthode casesAccessible en lui donnant casesVu et la copie de la grille avec le mur a
|   | tester dedans et on stocke le résultat dans le tableau d'ArrayList de String resultatTout
|   | On récupère la liste caseVu
|   | SI (casesVu n'est pas de la même taille que nbCasesAVisiter)
|   |   | On écrit un message d'erreur
|   | On return le boolean de si (casesVu n'est pas de la même taille que nbCasesAVisiter)
```

FIN

Chemin optimisé allant d'un joueur vers son objectif le plus proche

DEBUT

```
| On crée une String départ avec les coordonnées du joueur ayant appelé la méthode
| On crée l'ArrayList caseDepartAlgo
| On crée l'ArrayList chemin
| On met la case de départ dans caseDepartAlgo
| On crée le tableau d'ArrayList de String resultatTout initialisé avec la méthode casesAccessible
| On stocke les casesVu
| On stocke les casePere
| FOR ( chaque case de casesVu)
|   | On crée un tableau d'integer de taille 2
|   | on instance avec les coordonnées de la caseVu en cours
|   | SI ( la coordonée Y de caseCheck est égal à la ligne cible du joueur ayant appelé)
|   |   | on ajoute la case à chemin
|   |   | on ajoute son pere
|   |   | Le nouveau pere est la caseVu actuelle
|   |   | WHILE ( que la dernière case du chemin n'est pas la case de l'emplacement du
|   |   |   | joueur actuel )
|   |   |   | FOR ( toutes les cases de casesVu à
|   |   |   |   | SI (casesVu equals pere)
|   |   |   |   |   | On ajoute la caseVu actuelle au chemin
|   |   |   |   |   | Le pere est le pere de la case qu'on vient d'add
|   |   |   | break
|   | on retourne chemin
```

FIN

Algorithme pour le BFS

DEBUT

```
| Créer une liste vide appelée openList
| Créer une liste vide appelée closedList
| Créer une map appelée parentMap pour stocker les relations parent-enfant
| Ajouter le nœud de départ à openList et closedList
| Initialiser parentMap avec le nœud de départ associé à null
| Tant que openList n'est pas vide
|   | Sélectionner le nœud actuel de openList (premier élément)
|   | Retirer le nœud actuel de openList
|   | Pour chaque nœud dans la liste des objectifs
```

```

|   |   |   Si le nœud actuel est équivalent au nœud objectif
|   |   |       Construire le chemin en remontant les parents à partir du nœud actuel
|   |   |       Afficher le chemin trouvé
|   |   |       On retourne le chemin
|   |   Fin Si
|   |   Fin Pour
|   |   Ajouter le nœud actuel à closedList
|   |   Obtenir les voisins du nœud actuel dans la grille
|   |   Pour chaque voisin parmi les voisins
|   |       Si le voisin n'est pas null
|   |           Si le voisin n'est ni dans openList ni dans closedList
|   |               Ajouter le voisin à openList
|   |               Mettre à jour parentMap avec le nœud actuel comme parent du voisin
|   |           Fin Si
|   |   Fin Si
|   |   Fin Pour
|   Fin Tant que
|   On retourne null si le chemin n'est pas trouvé

```

FIN

V. Les difficultés rencontrées

1. Penser le mode de fonctionnement du jeu

En nous focalisant de plus près sur le jeu, nous nous sommes confrontés à la difficulté de sa représentation. En effet, la particularité du jeu est qu'il y a deux actions possibles qui sont à prendre en compte, le déplacement du pion et la pose des murs. Ces actions doivent se faire sur une même grille pourtant.

Après de longues réflexions et plusieurs hypothèses d'interfaces, la solution qui nous semblait la plus adéquate était de considérer que la pose de murs et le déplacement du pion devait se faire sur deux sous-grilles » distinctes de la grille du jeu. Pour cela, nous avons décidé que toutes les cases « cliquables » pour un déplacement de pion ou la pose d'un mur, correspondent respectivement à toutes les cases ayant des coordonnées paires et celles ayant des coordonnées impaires.

La case « cliquable » pour la pose d'un mur doit être son centre (un mur se dresse sur trois cases de la grille donc son centre est la case du milieu).

Finalement nous sommes restés sur ce principe de jeu mais avec une interface générée en forme terminal. Les cases « cliquables » (centre du mur à placer et case où l'on souhaite déplacer le pion) deviennent donc des coordonnées de la grille du jeu saisies par les joueurs.

2. Implémentation d'une recherche heuristique

A la base, nous avons pensé que le jeu était tout à fait en cohérence avec les notions du cours du moment car étant un jeu qui peut se jouer sur une grille et dont l'un des principes est de chercher un chemin pour atteindre un emplacement sur le plateau de jeu, nous avons naturellement fait le lien avec les recherches heuristiques.

Notre idée était d'implémenter un algorithme A* sur le jeu. Cependant, la première difficulté était de penser l'algorithme sur un environnement dynamique. En effet, le fait qu'un mur puisse être posé de manière dynamique en guise d'obstacle au chemin du pion vers son objectif, nous pousse à redéfinir le coût à chaque fois qu'un mur est ajouté sur la grille car la distance estimée par rapport à l'objectif devra varier en fonction de cela.

L'objectif d'ailleurs, nous pose un autre problème dans ce jeu car habituellement, dans un jeu à grille comme celui-ci, nous cherchons à atteindre une case. Or l'objectif est ici beaucoup plus général car on cherche à atteindre toute une ligne, ce qui veut dire qu'on aurait non une seule case, mais plusieurs cases (toutes les cases de la ligne à atteindre) considérées comme l'objectif.

3. Toujours permettre au pion d'avoir une issue vers une des cases de la ligne à atteindre.

Dans ce jeu, le pion ne doit jamais être isolé (clôturé) par les murs et les bordures du plateau de jeu.

Cette partie était l'une des parties les plus difficiles à traiter de notre jeu. Si de base, elle devait être traitée comme étant un des fonctionnements normaux du jeu. Tout l'enjeu de cette partie fait finalement l'implémentation de notre IA et nous a demandé énormément de temps de focalisation et de traitement.

4. Implémentation d'une IA finalement basée le BFS et recherche du chemin optimisé pour le joueur.

Malgré qu'on n'ait pas pu appliquer l'algorithme A* qu'on a voulu au départ, nous avons surmonté le problème avec un algorithme qui permet d'avoir le chemin le plus optimisé pour le joueur partant de la case où se trouve son pion vers la ligne d'arrivée. L'algorithme est expliqué plus haut (4.3 Chemin optimisé allant d'un joueur vers son objectif le plus proche [ici](#)). Concernant le joueur artificiel pouvant jouer sur l'interface de notre jeu, l'algorithme appliqué est le parcours en largeur d'abord (BFS) avec une heuristique, donc la Best First Search. Il nous fallait en effet pouvoir absolument appliquer une simulation automatique sur notre interface graphique et le BFS était le plus approprié à ce stade pour nous mais si on aurait voulu y mettre notre algorithme personnalisé.

VI. Conclusions

Pour conclure, notre Quoridor fonctionne avec une interface graphique et deux types d'IA, la première est une recherche du chemin optimal allant d'un point de départ vers la ligne d'arrivée, elle fonctionne sur le main de la classe Jeu en forme console.

La deuxième IA, qui fait l'objet de notre simulation automatique d'action est basée sur le BFS, c'est celle qui est représentée sur notre interface graphique.

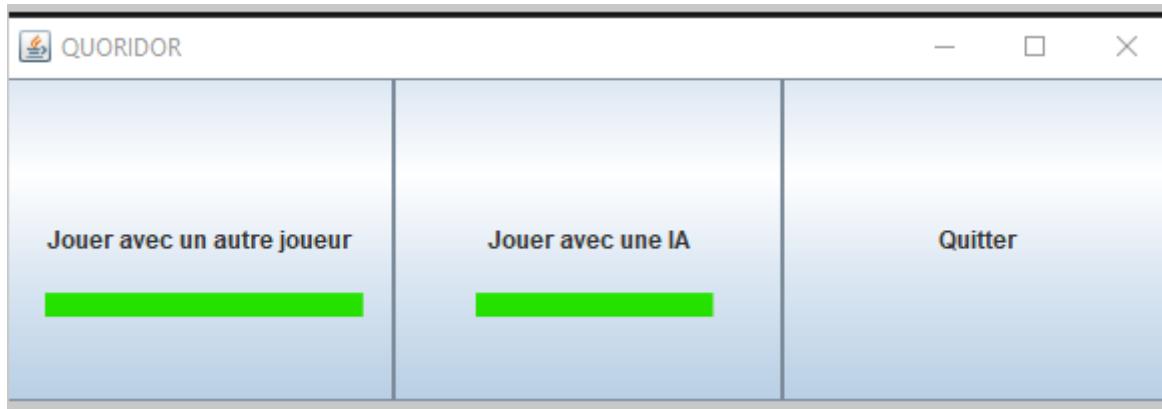
Nous avons alors pu permettre une interface proposant deux modes de jeu, le premier se jouant « joueur contre joueur » et le second « joueur contre joueur artificiel ». Les statiques d'évaluation n'ont pas pu être mesurées dans les derniers temps qui nous sont impartis.

Nous sommes aussi conscient de toutes les améliorations possibles sur le jeu et de la manière dont la partie intelligence artificielle pourrait être mieux exploitée. Nous en avons beaucoup appris que ce soit sur le projet ou sur notre méthode d'organisation de travail.

En sommes, ce fut un projet très enrichissant et un vrai défi pour chacun de nous dès la conceptualisation jusqu'à sa production finale même si elle n'est pas parfaite.

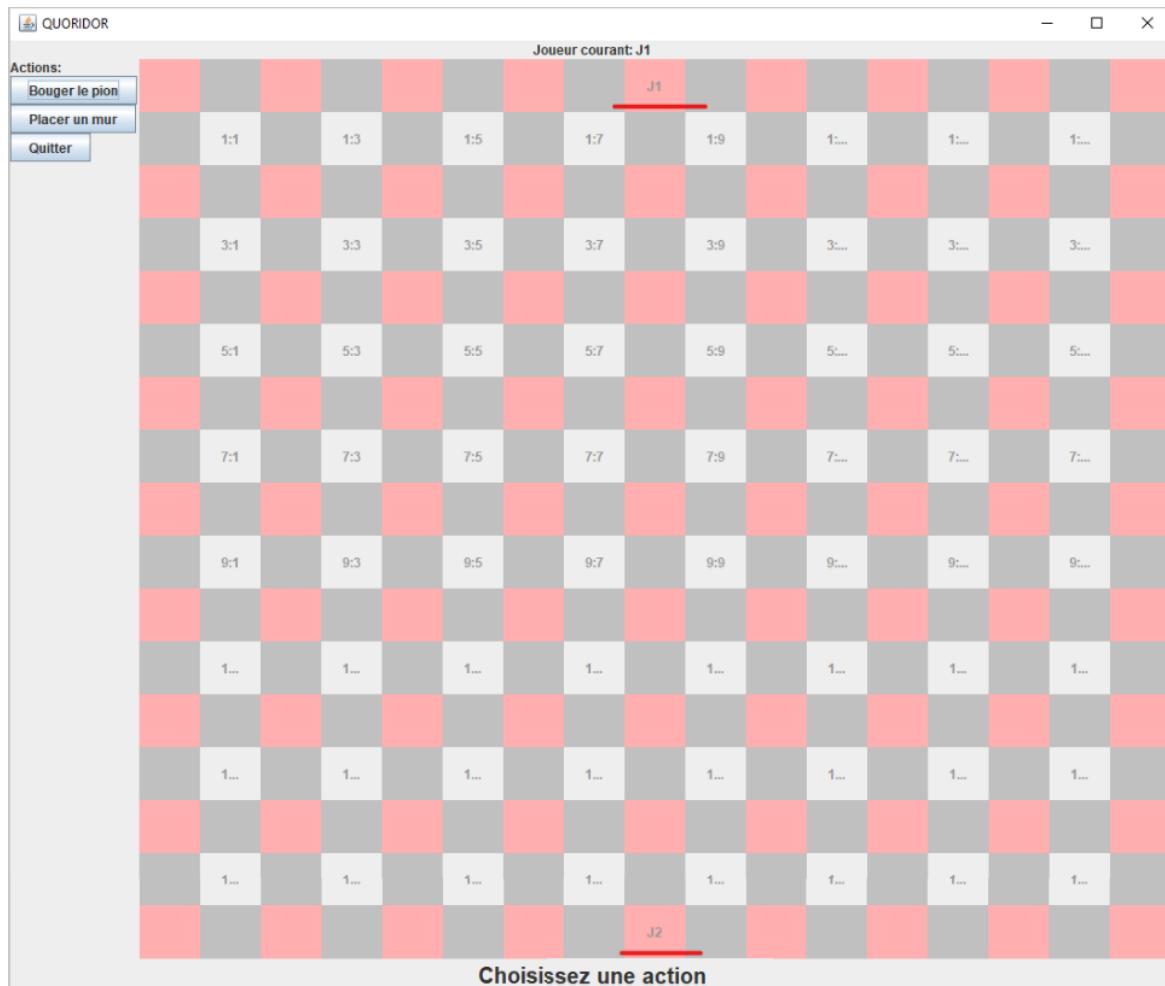
VII. Un guide rapide d'utilisation de l'interface

→ Notre interface présente deux choix de mode de jeu :



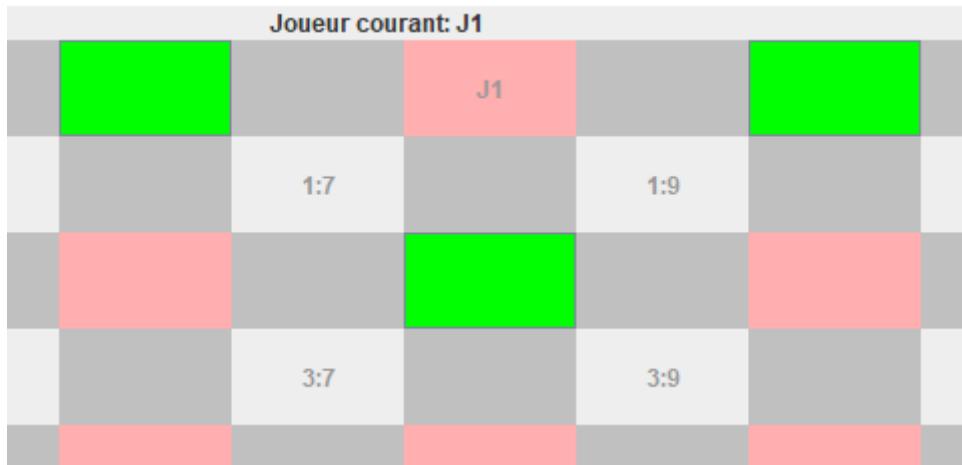
Il suffit de cliquer sur le mode souhaité pour commencer une partie.

→ Une fois qu'un mode de jeu est choisi, vous atterrissez sur l'interface de jeu avec à gauche les boutons d'action pour choisir si vous souhaitez placer un mur ou bouger votre pion (représenté par J1 ou J2 sur la grille de jeu).

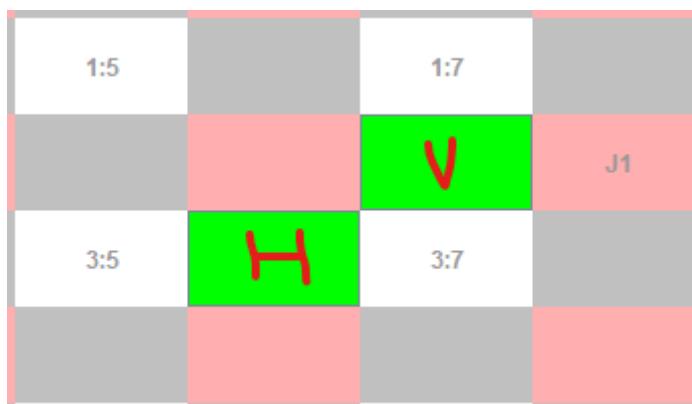


Au bas de l'interface, une indication vous est proposée pour les actions que vous devez faire.

- ➔ Lorsque vous avez choisi une action, "bouger le pion" par exemple, les cases sur lesquelles vous pouvez déplacer votre pion sont mises en vert.



- ➔ Si vous choisissez l'action "Placer un mur", les cases cliquables se montrent en vert, elles correspondent aux centres des murs (un mur se compose de 3 cases). Cliquez une première fois sur la case où vous voulez placer votre mur, il vous sera ensuite proposée de choisir l'orientation du mur (voir image ci-dessous).



H → Horizontale

V → Verticale

Dans le mode "Jouer avec une IA", l'IA correspond à J2, elle ne peut pas placer de mur mais se déplace seulement. Vous pouvez en revanche faire les deux actions.

- ➔ Vous pouvez quitter le jeu à tout moment via le bouton "Quitter".

VIII. Annexe

5. Historique d'organisation

[Réunions projet IA - Google Docs](#)

6. Idées d'interfaces du jeu

[Quoridor](#)

7. Notre idée de solution pour A*

Notre solution, mais qu'on n'a pas pu mettre en œuvre était ainsi :

On attribue un coût dans toutes les cases dans lesquelles le pion peut se déplacer, c'est-à-dire toutes les cases de coordonnées paires.

Dans la logique de l'algorithme A*, ce coût devra correspondre à l'addition de g (la distance réelle de la case courante par rapport à la case de départ) avec h (la distance estimée de la case courante par rapport à la case d'arrivée).

La distance qui sépare chaque case devra être la même soit 1. De ce fait, g est incrémentée de 1 pour chaque case en partant de la case de départ. En ce qui concerne h , on y fixe une constante différente en fonction de la ligne où se trouve la case (exemple : si la case est dans une ligne éloignée de la ligne à atteindre, h sera plutôt grande et inversement h sera plutôt petite).

On récupère toutes les cases adjacentes à la case courante (cases explorables) et parmi celles-ci on récupère celles ayant le coût le moins élevé. Si une case se trouvant dans la collection de toutes les cases explorables a un coût plus petit que ceux des cases adjacentes à la case courante alors cette case devient la case courante.

On trouve le chemin quand la case courante est une case d'arrivée.

Or, nous avons plusieurs cases considérées comme cases d'arrivée (la ligne de départ du joueur adverse). L'idée est alors de mettre toutes les cases de cette ligne dans un tableau et l'on considère qu'on a trouvé un chemin si la ligne d'arrivée contient la case courante.

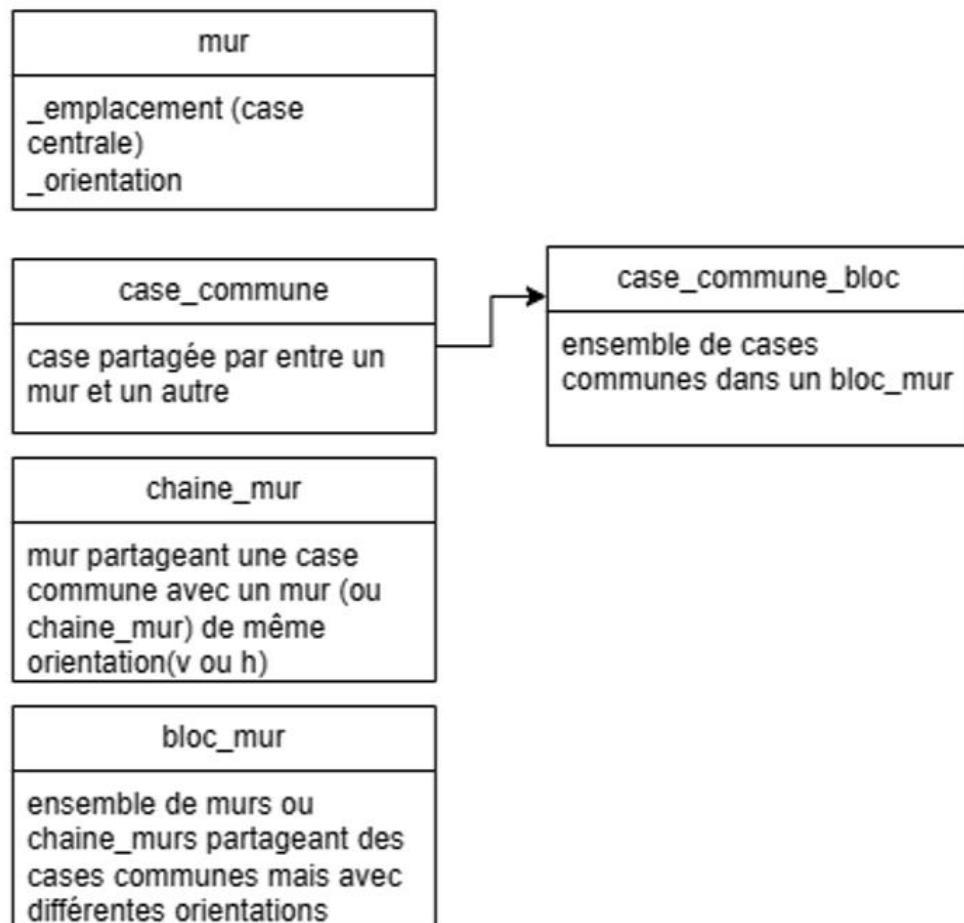
Comme le jeu est dynamique, on ne pourra pas garder une même case comme case de départ tout au long du jeu, on devra donc redéfinir la case de départ chaque fois que le dernier chemin trouvé sera bloqué par un mur. Cette redéfinition de la case de départ va réévaluer les coûts de toutes les cases et A* retourne un nouveau chemin partant de la nouvelle case de départ (case courante). L'algorithme est alors relancé chaque fois que le tour de jeu change.

8. Aides pour l'algorithme A*

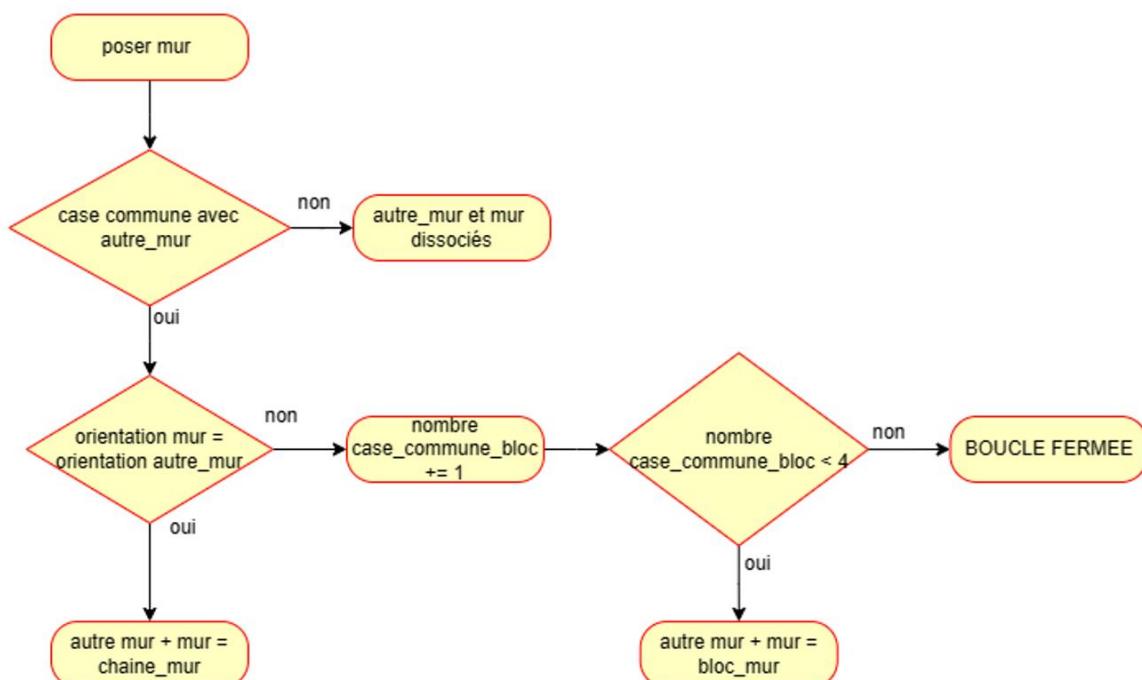
https://www.youtube.com/watch?v=-L-WgKMFuhE&ab_channel=SebastianLague

 [Algorithme A* : définition et explications \(techno-science.net\)](#)

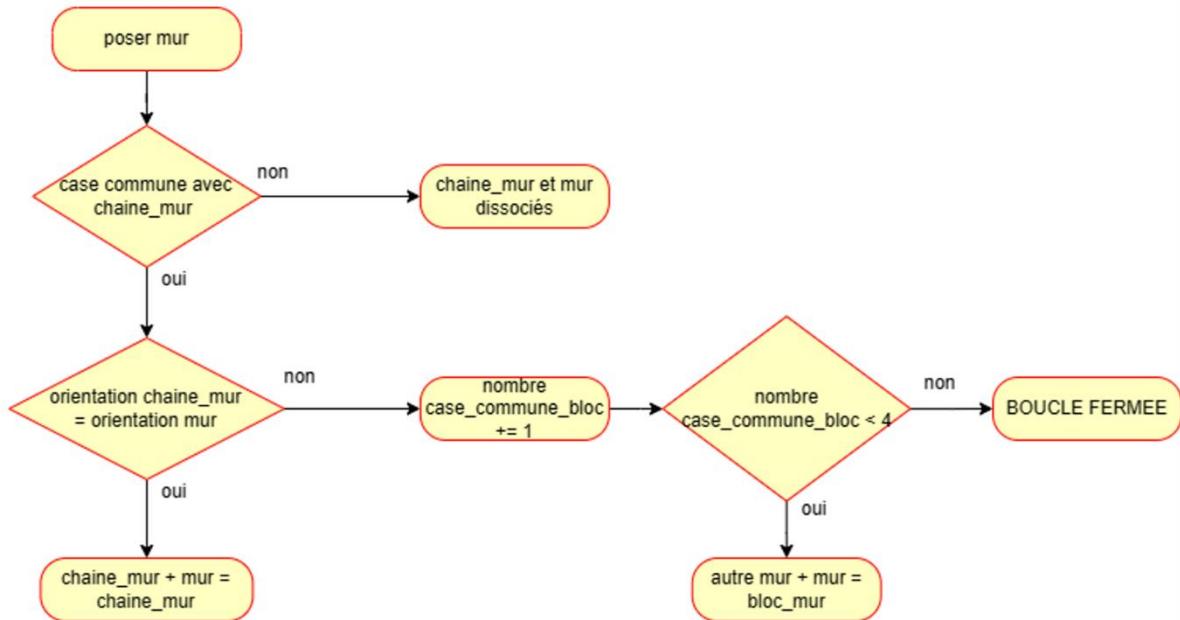
9. Modélisations d'une idée d'algorithme pour éviter l'isolation d'un joueur :



CREATION D'UNE CHAINE DE MURS ET D'UN BLOC DE MURS



CONTINUER UNE CHAINE DE MURS AVEC UNE CHAINE DEJA EXISTANTE



CONTINUER UN BLOC DE MURS AVEC UN BLOC DEJA EXISTANT

