Université de Namur

INFOB317: Techniques d'intelligence artificielle

Projet QuoridorIA groupe 1

Project workers :

Snickers Florent
Polet Simon
Leblanc Stéphane

Professeur:
Jean-Marie Jacquet

Assistant:
Manel Barkallah

Contents

1 Introduction

Ce document a pour but de présenter et développer les différents raisonnements appliqués pour construire une intelligence artificielle pouvant jouer au jeu Quoridor. En plus de cette intelligence, un bot a été créé pour répondre aux questions des joueurs. Ces deux parties ont été développées en Prolog.

Pour une meilleure expérience de jeu, nous devions lier le tout à une interface web. Malheureusement, nous avons recontré pas mal de problèmes. Nous avons réussi à lier complétement le bot, et l'ia pour les deux premiers coups du jeu mais le reste ne fonctionnait pas bien. Dès lors, nous vous avons join seulement les fichiers Prolog. Nous sommes désolés pour cela mais nous ne doutons pas de la qualité de notre bot et intelligence artificielle !

Pour ce travail, nous avons considéré la version à 4 joueurs du jeu. Deux pions étant controlés par des joueurs et deux pions étant controlés par une intelligence artificielle.

2 Bot

Pour construire un bot pouvant répondre aux questions des joueurs, nous nous sommes basés sur le fichier *qbot-elm.pl* proposé par Monsieur Jacquet. Ce bot se base sur des mots clés pour identifier la phrase de l'utilisateur.

Chaque input de l'utilisateur est d'abord transformé en une liste, où un élément de celle-ci correspond à un mot de l'input. Grâce à cela, il est facile d'analyser chaque mot.

Ensuite, sur base de mots clés choisis et considérés comme pertinents par notre groupe, le bot analyse chaque mot de l'input avec chaque mot clé défini. Cela peut se faire grâce au prédicat **isSimilarMclef**. Celui-ci utilise le prédicat **isub** qui mesure la distance entre deux mots. Dès lors, nous considérons **IsSimilarMclef** comme vrai, si la distance entre un mot de l'input et un mot clé est strictement plus grande que le coefficient 0.93. Si nous utilisions simplement **Member**, une simple faute de frappe n'aurait pas permis au bot de reconnaître le mot clé.

Ensuite, si le bot a pu identifier un mot clé dans la phrase de l'utilisateur, l'intrepréteur Prolog va tenter d'unifier une **règle_rep** qui associe à un mot clé un pattern. Dès lors, nous savons que nous pouvons comparer ce pattern avec l'input de l'utilisateur. Nous considérons que l'input match avec ce pattern si il existe au moins deux mots de l'input qui sont similaires aux mots du pattern. Cela peut encore se faire grace au prédicat **isub**. Si c'est le cas, nous pouvons fournir la réponse incluse par **regle_rep** à l'utilisateur car son entrée est assez proche d'un pattern pour donner une réponse.

Enfin, dans le but de fournir deux réponses différentes si l'utilisateur demande deux fois la même chose, nous avons utiliser les **memory_file**. Cela nous sert simplement de buffer pour comparer l'input actuel avec l'input précédent. Si les deux sont les mêmes, le bot donnera la réponse ayant le poids le plus faible (deuxième argument de **regle_rep**). Sinon, il donnera une réponse ayant le poids le plus fort.

Par manque de temps, nous avons pas implémenter le fait que le bot puisse donner un coup à faire à l'utilisateur. Néanmoins, vu note IA, cela ne serait pas compliqué. Il suffirait de récupérer l'état actuel du jeu et appeler le prédicat **strat** qui fournira une action à faire.

3 IA Quoridor

Dans cette dernière partie du projet, nous avons implémenté une intelligence artificielle en prolog. Celle-ci est capable de déterminer les prochains coups qu'elle effectuera à partir d'un état donné du jeu. Cette intelligence artificielle a été implémentée en suivant des algorithmes de la théorie des jeux ainsi que des méthodes de recherche.

3.1 Stratégie Générale D'Implémentation

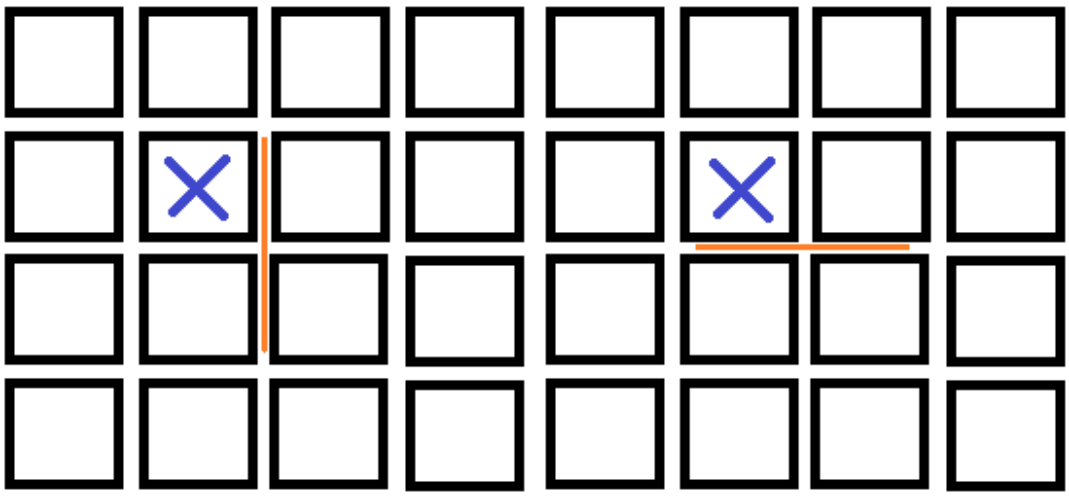
Notre objecctif a été d'implémenter un algorithme Minimax avec élagage alpha-beta afin de rechercher la meilleure action possible à effectuer à un état donné de la partie grâce à une valeur calculée sur les états terminaux de notre arbre et transmis aux parents.

3.2 Implémentation d'un espace d'état

Pour pouvoir utiliser un algorithme Minimax, nous avons du commencer par définir ce qu'était un état valide pour le jeu. Nous avons défini un état du jeu selon la structure suivante :

```
state(color(COLOR),
        players([player(color(bleu),N,pos([POSX,POSY])),
                player(color(rouge),N,pos([POSX,POSY])),
                player(color(vert),N,pos([POSX,POSY])),
                player(color(jaune),N,pos([POSX,POSY]))
        ],
        barriers([barrier(pos(POSX,POSY),plan(DISPOSITION),
                barrier(pos(POSX,POSY),plan(DISPOSITION),
                ...
        ])))
```

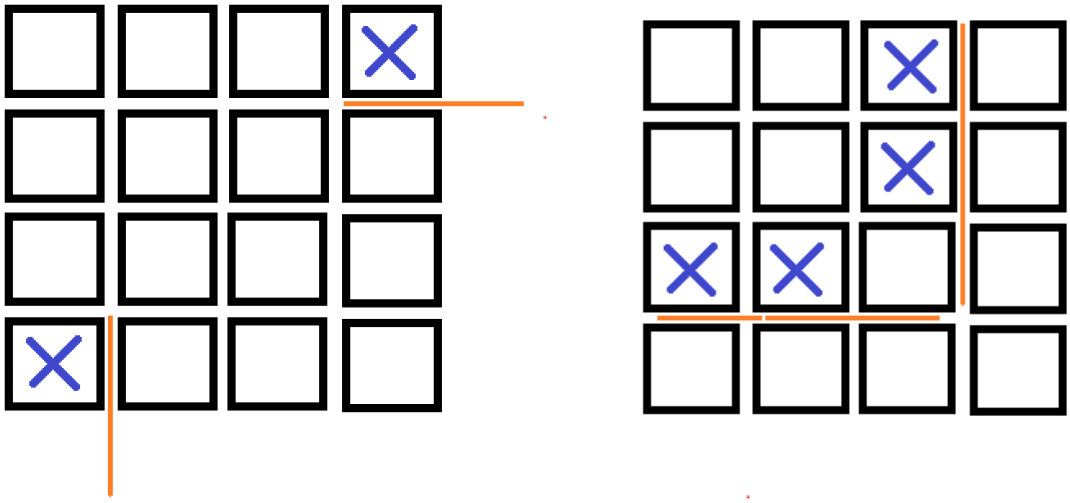
Un état possède donc une couleur (celle du joueur qui doit effectuer la prochaine action), un ensemble de joueurs (les 4 joueurs du jeu) et un ensemble de barrières (les barrières déjà déposées sur le jeu). Un joueur possède une couleur (la couleur le définissant), un nombre de barrières (le nombre de barrières restant à sa disposition) et une position (sa position actuelle sur le plateau de jeu). Une barrière possède une position (la case supérieure gauche des 4 cases qu'elle touche) et une disposition dans le plan (horizontale ou verticale).



(a) Représentation d'une barrière verticale (b) Représentation d'une barrière horizontale

Nous avons, dès leur définition, vérifié la validité de chaque sous structure de l'état. Pour ce faire, nous leur avons mis des contraintes dont en voici une liste non-exhaustive :

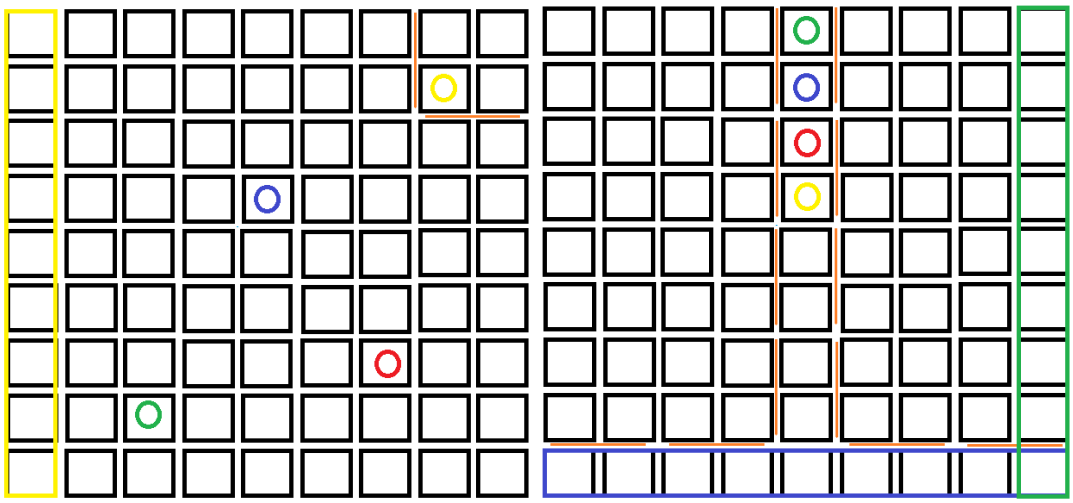
- Un joueur doit se trouver dans les limites du terrain.
- Une barrière doit se trouver dans les limites du terrain et ne doit pas dépasser du terrain.
- Une barrière de l'ensemble des barrières sur le terrain ne doit pas entrer en collision avec une autre.
- Un joueur doit avoir un nombre de barrières strictement positif.



(a) Problème des barrières hors-limites (b) Problème de la collision de barrières

3.3 Plus court chemin et deadlock ou livelock

Après avoir vérifié des contraintes relativement simples, nous nous sommes retrouvés face à une contrainte plus complexe à vérifier. Un état ne peut être valide que si aucun joueur ne se retrouve bloqué. Ce genre de situation peut provoquer un deadlock où le joueur n'a plus aucune possibilité d'action ou un livelock où le joueur peut jouer mais n'atteindra jamais son objectif. La difficulté de cette contrainte vient du nombre de cas aussi variés qu'exotiques dans lequels ces problèmes peuvent arriver. Par exemple, les exemples suivants :

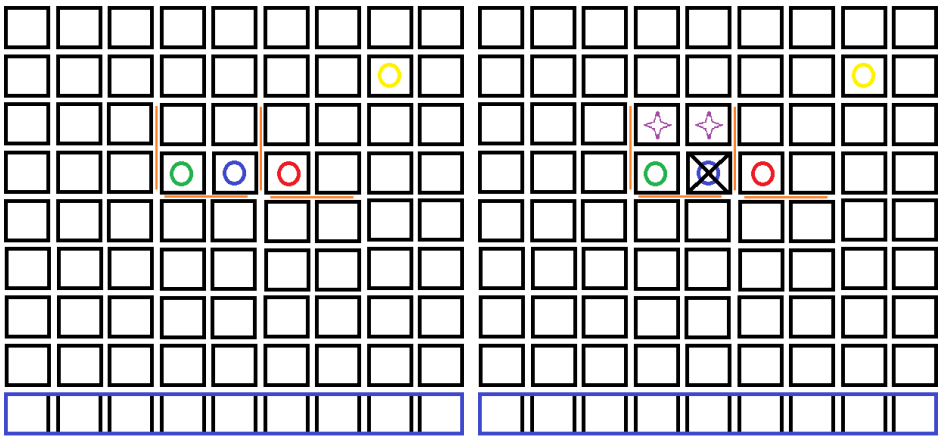


(a) Livelock simple (b) Deadlock complexe

Pour gérer tous ces cas de figure, nous avons pensé à une heuristique. Un état est valide si pour chacun des joueurs, il existe un chemin (plus court chemin) le menant à son arrivée. Nous avons donc implémenté un algorithme du plus court chemin qui, lorsqu'il ne trouve pas de solution, prouve qu'un état n'est pas valide.

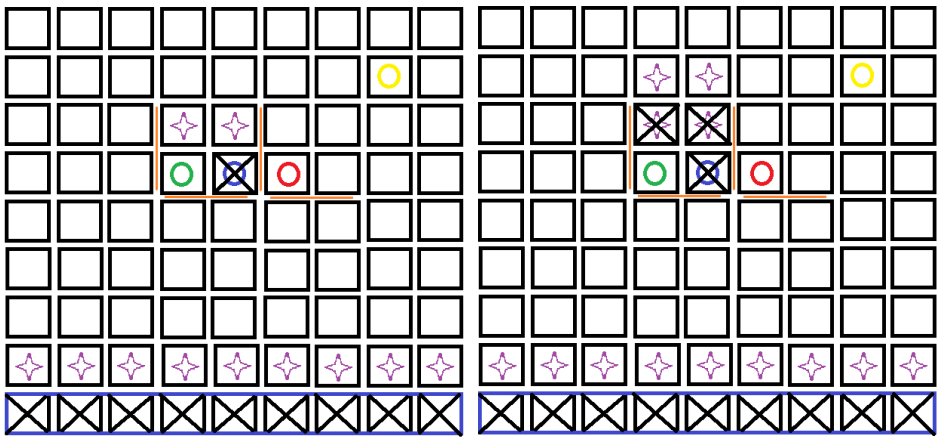
Afin d'avoir un algorithme efficace, nous avons implémenté un algorithme du plus court chemin grâce à une recherche bidirectionnelle. Cet algorithme utilise la méthode de recherche dans un arbre en largeur d'abord. Ci-dessous sont expliqués les pas de l'algorithme mis en situation (appliqué au joueur bleu) dans un cas relativement complexe :

- On démarre de la position de notre pion et on calcule toutes les positions accessibles. Ici, des barrières bloquent des positions adjacentes. Un joueur se trouve sur une position adjacente à notre pion donc on peut sauter par-dessus. Une barrière bloque la position derrière lui et une autre la position à sa gauche (vu par notre joueur). On peut donc ajouter aux ouverts(1) les deux positions sur lesquelles se trouve une étoile mauve. On ajoute aux fermés(1) la position de notre pion.[1]



(a) État passé à la fontion (b) Successeurs côté joueur

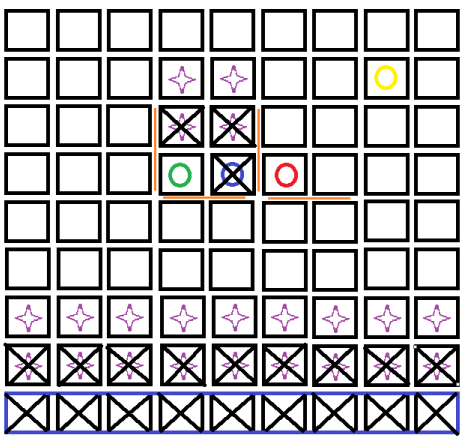
- Ensuite, on calcule les positions successeurs (positions accessibles à partir d'un ensemble de positions) de l'arrivée (arrivée du point de vue de notre joueur). On ajoute aux ouverts(2) les positions calculées et aux fermés(2) les positions parents des successeurs calculés[2]



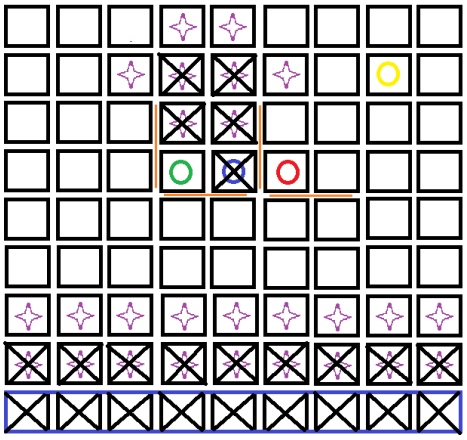
(a) Successeurs côté arrivée (b) Successeurs côté joueur

[1] ouvert(1) et fermé(1) désignent les ouverts et fermés calculés en partant de la position du pion
[2] ouvert(2) et fermé(2) désignent les ouverts et fermés calculés en partant de l'arrivée

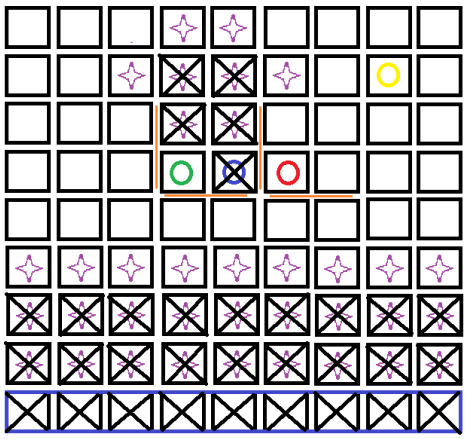
- *On répète ensuite les mêmes étapes jusqu'à arriver à un cas de base.*



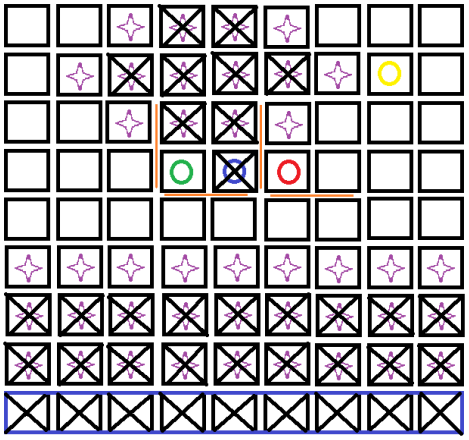
(a) Successeur côté arrivée



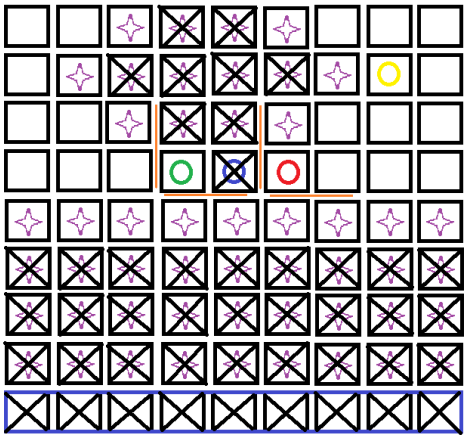
(b) Successeur côté joueur



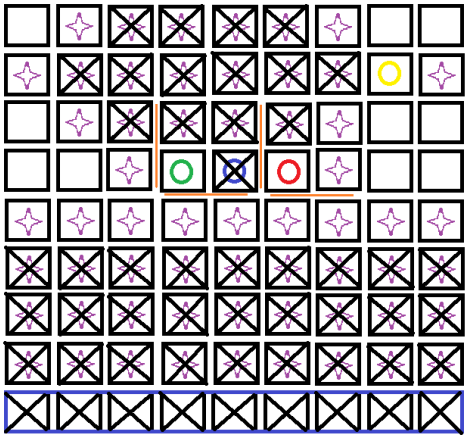
(a) Successeur côté arrivée



(b) Successeur côté joueur

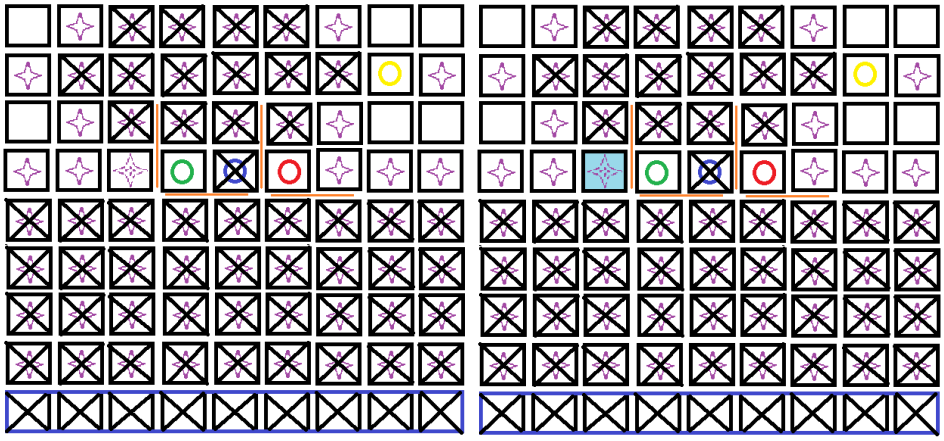


(a) Successeur côté arrivée



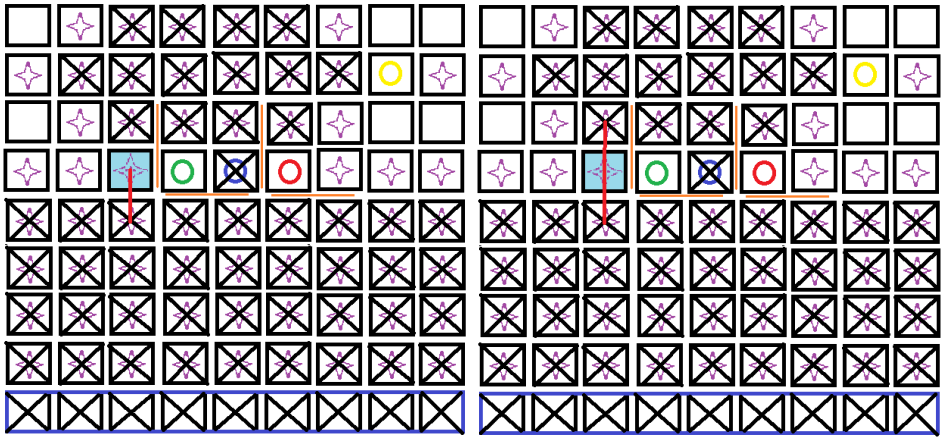
(b) Successeur côté joueur

- L'algorithme trouve une intersection entre l'ensemble des ouverts(1) et l'ensemble des ouverts(2). C'est notre cas de base.

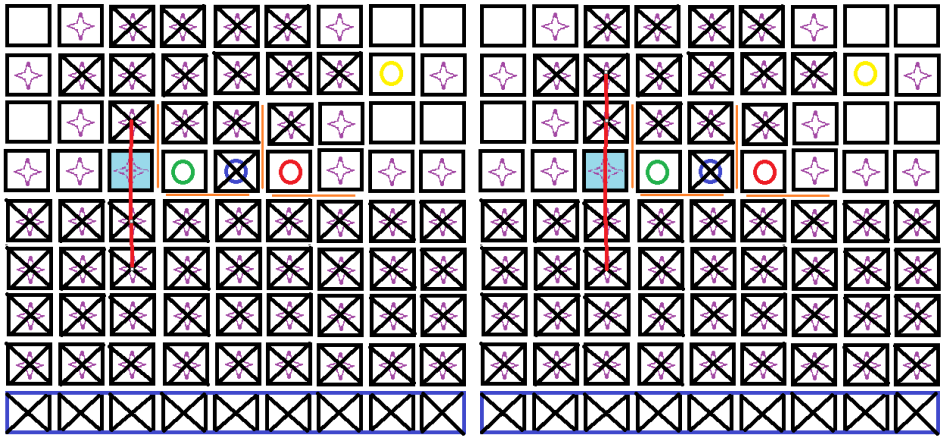


(a) Successeur côté arrivée (b) Successeur côté joueur

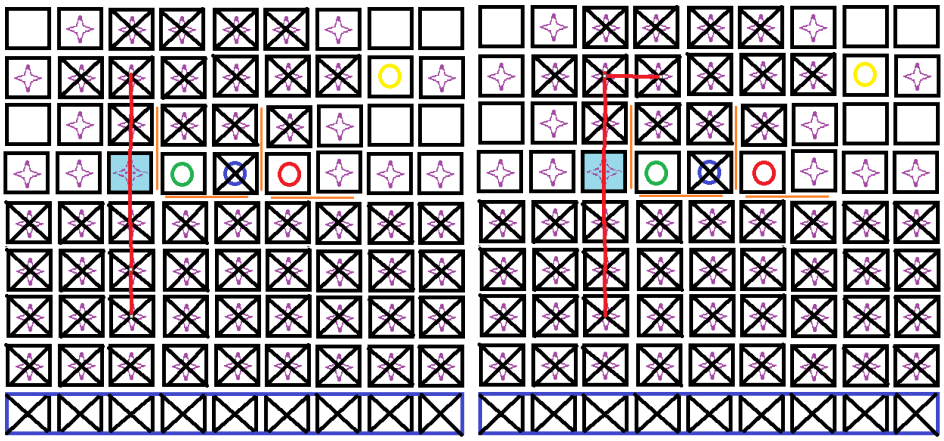
- À partir d'ici, on va pouvoir reconstruire le chemin précédemment emprunté grâce à des références entre enfants et parent créées lors de la recherche.



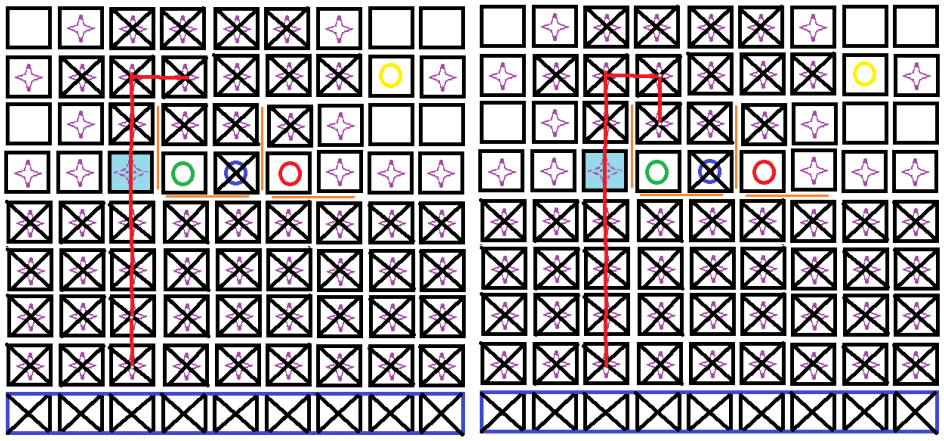
(a) Construction du chemin (b) Construction du chemin



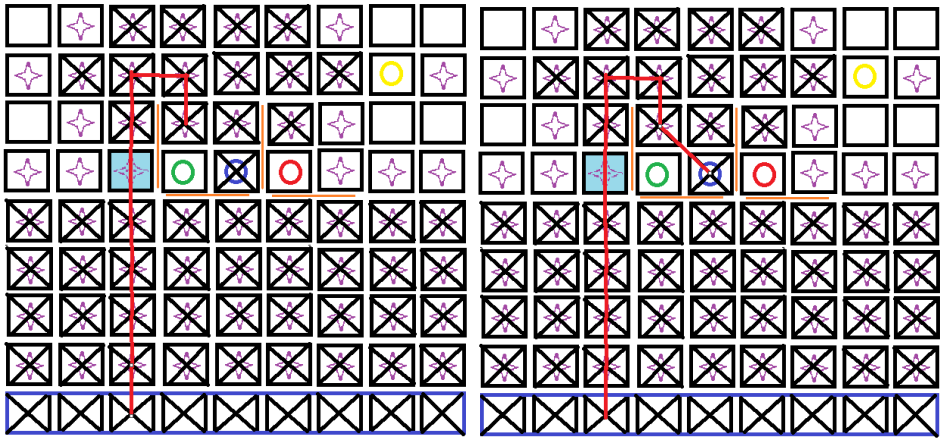
(a) Construction du chemin (b) Construction du chemin



(a) Construction du chemin (b) Construction du chemin



(a) Construction du chemin (b) Construction du chemin



(a) Construction du chemin (b) Construction du chemin

Nous avons défini une série de règles auxiliaires permettant de faire fonctionner *shortest_path/9*. Parmi celles-ci, les plus importantes sont *adjacent/2* qui calcule les positions adjacentes à une position ; *accessible/5* qui calcule les positions accessibles à partir d'une position donnée ; *successeur/8* qui applique *accessible/5* sur une liste de positions et crée des références entre les positions accessibles et les cases d'où elles ont été calculées.

3.4 Définitions des actions et des transitions entre états

L'étape suivante avant d'implémenter l'algorithme principal nécessaire à la stratégie de notre IA consiste à définir les actions possibles à partir d'un état ainsi que les transitions possibles. Fort heureusement, nous avons pu réutiliser des règles utilisées plus tôt afin de les implémenter. La règle *action/2* permet de déterminer les actions possibles à partir d'un état. Elle peut être utilisée grâce à des règles auxiliaires dont les plus importantes sont *move/2* et *drop/2*.

Les actions renvoyées par *action/2* ont la forme suivante :

- [act(move),COLOR,POSITION], si c'est une action de déplacement ;
- [act(drop),COLOR,#BARIIER,BARRIERS], si c'est une action de pose de barrière ;

Rem : COLOR est la couleur du joueur faisant l'action ; POSITION est la position d'arrivée du joueur, une fois l'action effectuée ; #BARRIER est le nombre de barrières restant au joueur après avoir posé sa barrière ; BARRIERS est la liste des barrières présentes sur le jeu lorsque la nouvelle barrière est posée.

Une fois que nous avons créé la règle action, nous avons poursuivi en créant la règle transition *trans/3* qui, à partir d'un état et d'une action[3], nous permet de récupérer l'état suivant. Elle peut être utilisée grâce à deux règles auxiliaires qui sont *trans_ move/3* et *trans_ drop/3*.

3.5 Rappel de la stratégie d'implémentation avant la dernière ligne droite

Si l'on veut représenter le jeu Quoridor sous forme d'arbre où les noeuds sont des états acceptables du jeu et les branches sont des transitions entre ces noeuds, on se confronte à un problème de taille. En effet, chaque noeud père peut posséder au maximum 132 noeuds fils (nombre de coups acceptables maximum à partir d'un état). Si ce nombre de fils reste calculable par un ordinateur, il suffit de quelques couches pour atteindre un nombre de petits-fils démesuré. De plus, dans Quoridor, on peut jouer des coups qui nous ramènent à un état atteint précédemment dans le jeu, rendant l'arbre des états successifs acceptables potentiellement infini.

Afin que l'IA calcule le coup suivant à utiliser dans un temps raisonnable, nous avons opté pour un algorithme Minimax dont la méthode de recherche est en longueur limitée d'abord.

En plus de ce qui a été expliqué plus haut, nous avons aussi implémenté ce qu'est un état initial et un état final. Donc, pour l'instant, nous avons ceci :

- S := state : Ensemble de toutes les configurations possibles du jeu (que l'on peut générer)
- P := list_player : Ensemble des joueurs (présents dans state)
- I := initial_state : état initial
- Term : S -> v,f := final_state : états finaux du programme
- Actions : S X P -> Act := action : action à effectuer
- Trans : S X A -> S := trans : transition entre deux états

Dès lors, ce qu'il nous reste à implémenter à partir d'ici sont les règles suivantes :

- Util : S X P -> R := util : calcule l'utilité d'une branche
- Strat : S -> A := strat : stratégie de l'IA

[3] L'action passée doit être valide pour l'état passé.

3.6 Définition de la règle d'utilité

Notre algorithme, pour calculer l'utilité d'un état, se base sur le jeu Quoridor à 4 joueurs. Cela implique que le résultat apporté par notre fonction d'utilité soit un vecteur à 4 dimensions.

<UtilPlayer1,UtilPlayer2,UtilPlayer3,UtilPlayer4>

Pour chaque joueur, l'utilité de l'état actuel du jeu va être calculée par une équation linéaire de la manière suivante:

$$UtilPlayer = C_1 * \frac{\#GPCC}{\#PCC + \#BPCC} + C_2 * (\frac{\sum_i (LNC_i - LPCC)}{\#PCC - 1} - LPCC + W) + C3 * \#B$$

Pour créer cette équation, nous sommes partis de 3 heuristiques que nous avons définies nous-mêmes.
1) Au moins nous avons de (bons) chemins vers l'arrivée, au plus l'utilité de l'état sera grande[4].
2) Au plus court est notre chemin vers l'arrivée, au plus l'utilité de l'état sera grande.
3) Au plus il nous reste de barrières, au plus l'utilité de l'état sera grande[5].

- #PCC *(Plus Court Chemin)* désigne le nombre de plus cours chemins disponibles à partir de la position d'un joueur.
- #GPCC *(Good Plus Court Chemin)* désigne parmi tous les plus courts chemins, ceux qui ont un nombre de mouvements suffisamment proche de #PCC.
- #BPCC *(Bad Plus Court Chemin)* désigne parmi tous les plus courts chemins, ceux qui ont un nombre de mouvements trop éloigné de #PCC.
- LPCC *(Length Plus Court Chemin)* désigne la longueur du plus court chemin.
- LNC_i *(Length Nearest Chemin)* désigne une à une les longueurs des plus courts chemins hormis le plus court chemin.
- #B *Barrier* désigne le nombre de barrières restant au joueur.

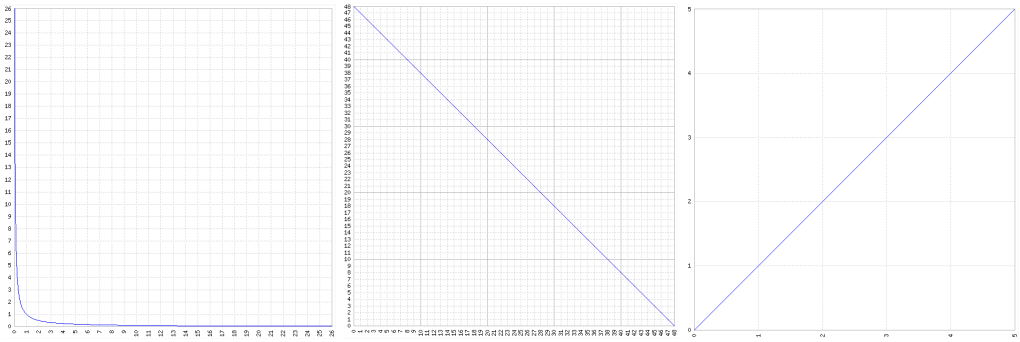
Rem : L'utilité calculée sur un état pour un joueur possède une valeur qui est toujours comprise entre 0 et 1[6].

[4]En effet, s'il nous reste peu de chemins, il est plus difficile pour l'adversaire de bloquer notre chemin le plus court et plus facile pour nous de s'assurer de pouvoir le prendre.

[5]En effet, s'il nous reste beaucoup de barrières, on peut bloquer l'adversaire ou sécuriser nos chemins.

[6]Le terme 1 est borné par 0 et 1 ; le terme 2 est borné par 0 et 48 ; le terme 3 est borné par 0 et 5. Ils sont tous normalisés et multipliés par le pourcentage de l'importance que nous leur apportons puis divisés par 100. Nous avons donc bien une somme des termes compris entre 0 et 1.

Le premier terme de cet algorithme a été pensé pour se comporter comme une fonction inverse (en ne prenant bien évidemment que des résultats positifs), c'est-à-dire au moins on a de plus courts chemins, au plus le score augmente rapidement. Le second terme a été pensé pour se comporter comme une fonction linéaire pour que notre score augmente de la même façon lorsqu'on se rapproche de l'arrivée, quel que soit notre emplacement. Le troisième terme augmentera lui aussi le score de manière linéaire ; au plus on a de barrières, au plus notre score augmentera.



(a) Tendance voulue pour le terme 1 (b) Tendance voulue pour le terme 2 (c) Tendance voulue pour le terme 3

Certaines parties de cet algoritme sont paramétrables à souhait pour améliorer les résultats que l'on obtient.

- C1, C2, C3 sont des constantes permettant de normaliser les termes de l'équation linéaire et de pondérer l'équation en fonction de l'importance que nous donnons à ceux-ci.[7]
- W est une constante que nous avons fixée à 48, le nombre maximum de cases pour un plus court chemin.[8]
- L est une constante que nous avons fixée à 6. Elle n'est pas présente dans la formule ci-dessus mais y intervient. L est le nombre de coups qui permet de déterminer si un chemin est bon ou mauvais en fonction du plus court chemin.[9]

Afin de pouvoir fonctionner, la fonction *util/2* a besoin d'un certain nombre de règles auxiliaires dont les principales sont *util_player/6* qui calcule l'utilité pour un joueur en particulier ; *number_path/7* qui calcule le nombre de plus courts chemins de notre joueur à l'arrivée ; *sum_of_difference_path_length/3* qui renvoie la somme de la différence entre les presque plus courts chemins et le plus court chemin ; *number_bad_path/4* qui calcule le nombre de mauvais plus courts chemins dans une liste de plus courts chemins.

[7] Après plusieurs essais, nous avons décidé de fixer C1 := 11.38 ; C2 := 1.54 ; C3 := 2.45. Cela correspond respectivement à 11.38 %, 73.92 %, 14.7 % de la valeur d'utilité.

[8] Cette constante sert à déplacer le graphe de l'utilité en fonction du deuxième terme pour rendre toute solution positive.

[9] Nous avons choisi la valeur 6 car c'est le nombre de coups qu'il y a entre le plus court chemin et le plus long pour un pion lorsqu'il n'y a pas d'obstacles.

3.7 Définition de la stratégie

Nous y voilà, la dernière étape pour que notre IA puisse faire des choix ! Il s'agit ici de rassembler tout ce qu'on a fait précédement et de l'utiliser à bon escient dans notre règle *strat/2*. Nous avons commencé par implémenter cette fonction en suivant l'algorithme Minimax dans lequel nous effectuons une recherche en profondeur limitée d'abord.

- L'algorithme commence à la racine de l'arbre (l'état courant du jeu) où la profondeur est de 0.

Figure 16: Racine de l'arbre

- L'agorithme descend dans l'arbre, jusqu'à atteindre la profondeur p que nous définissons.

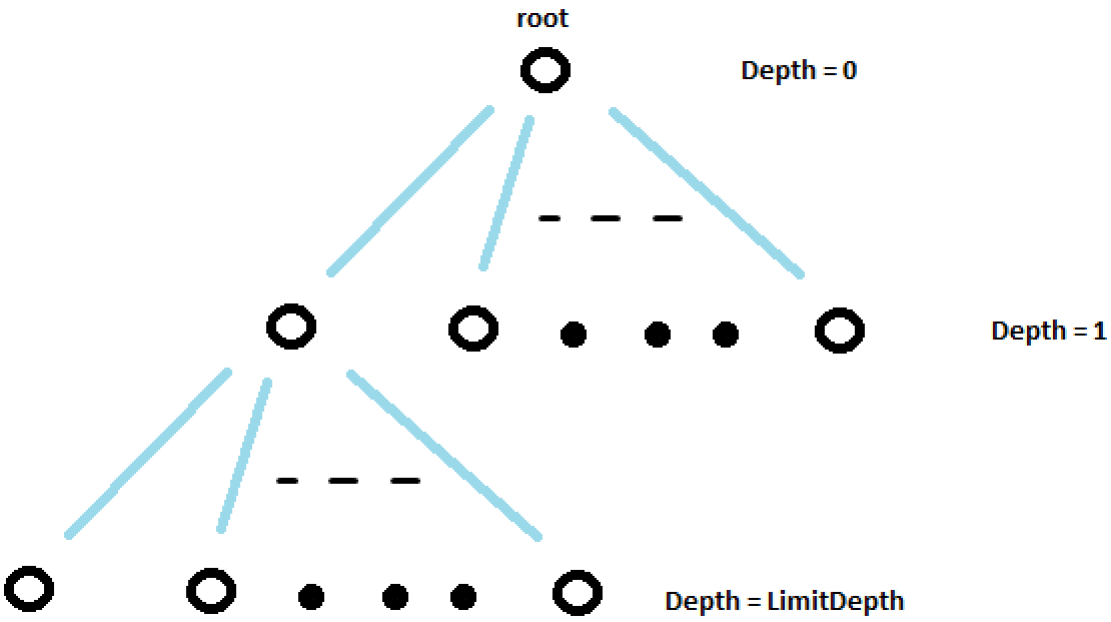


Figure 17: Calcul des fils du premier fils

- L'algorithme descend dans l'arbre jusqu'à atteindre la profondeur limite que nous avons définie.

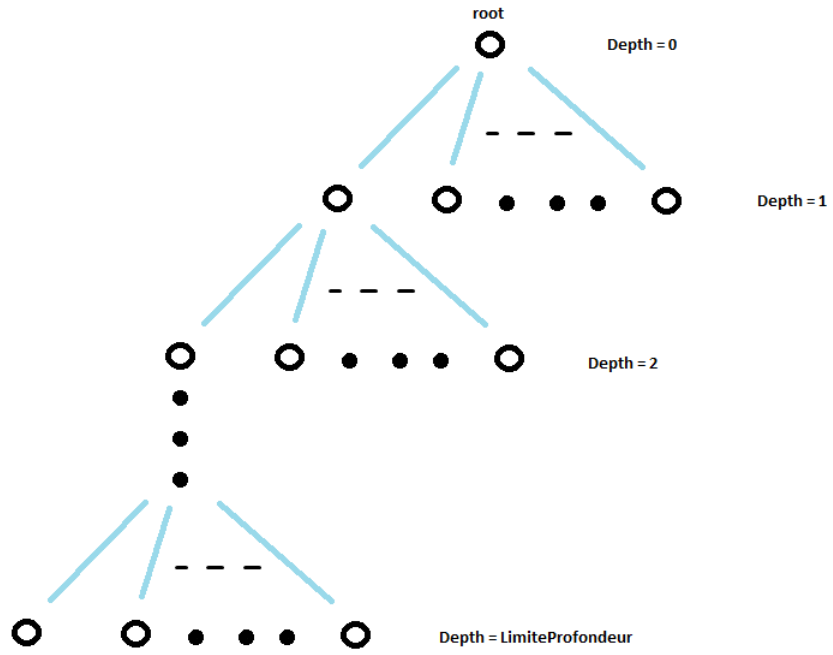


Figure 18: Calcul récursif jusqu'à atteindre la profondeur limite

- Une fois arrivé sur la dernière couche, l'algorithme calcule l'utilité d'un état et renvoie son vecteur.

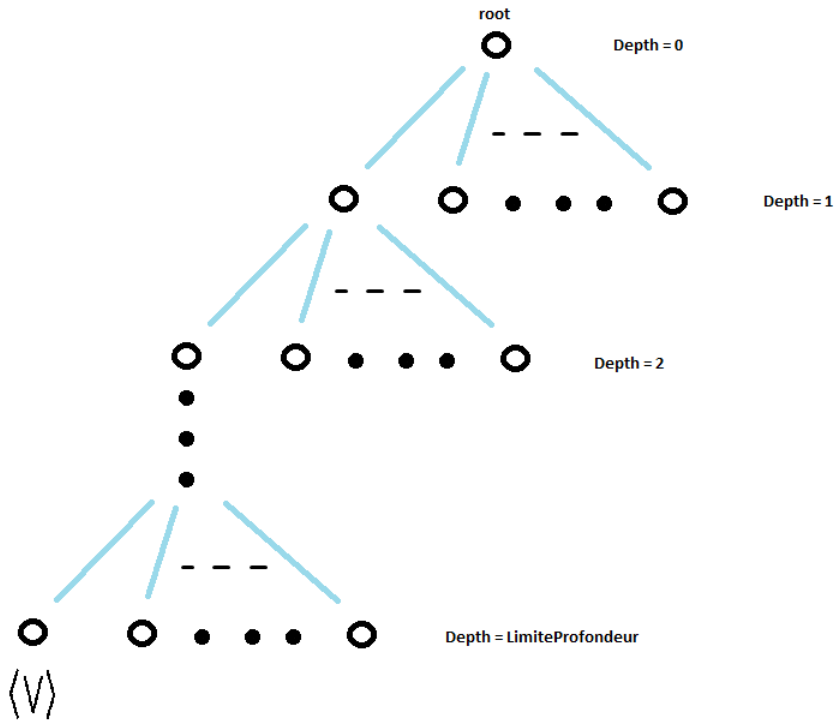


Figure 19: Calcul de l'utilité d'un état sur la dernière couche

- L'algorithme calcule 1 à 1 les valeurs des noeuds de la couche actuelle (la plus profonde ici).

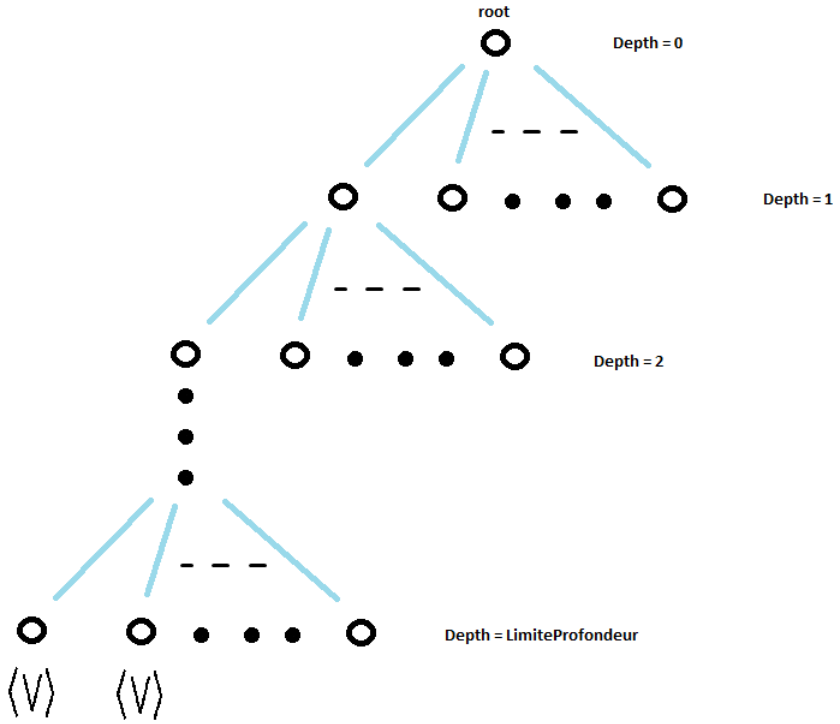


Figure 20: Calcul récursif des utilités des différents états

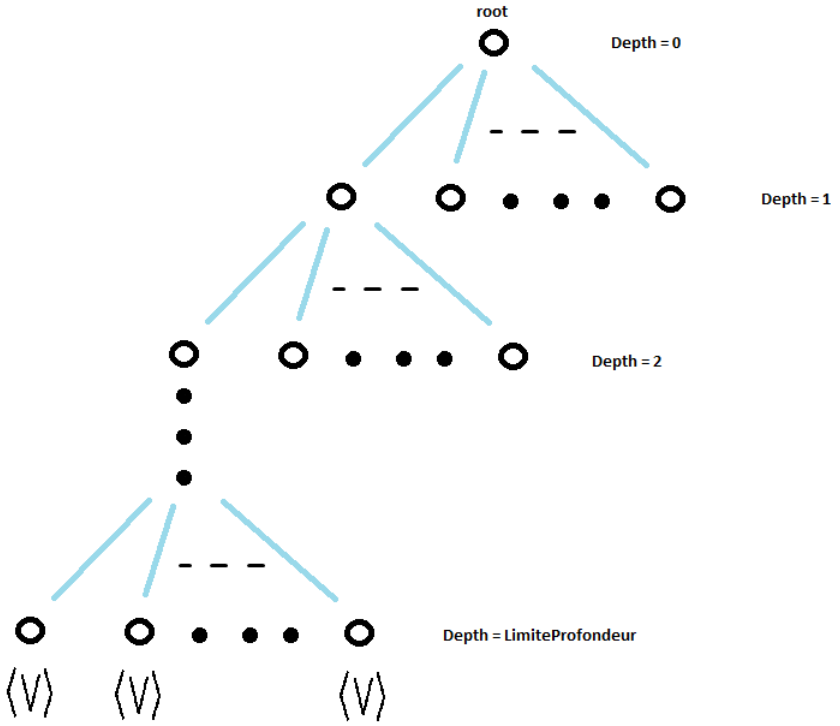


Figure 21: Calcul récursif des utilités des différents états

- L'algorithme retourne à la couche précédente avec le meilleur vecteur d'utilité si c'était à l'IA de jouer chez le parent, sinon il retourne avec le pire vecteur. Cette manière de procéder est connue sous le nom d'algorithme de Paranoïde où l'on considère que tous les adversaires s'allient contre nous et génèreront donc le pire vecteur pour nous. [10]

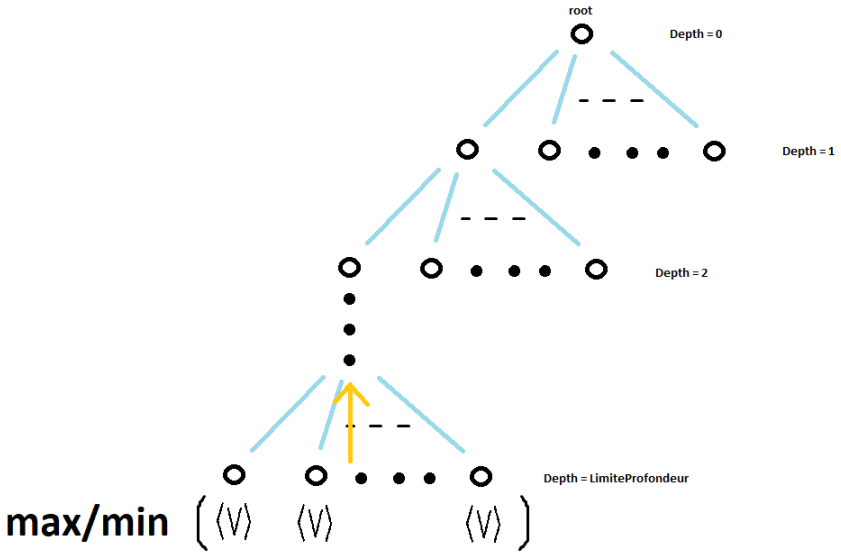


Figure 22: Utilisation de l'algorithme de Paranoïde pour le choix du meilleur vecteur

- Le meilleur ou le pire vecteur étant renvoyé au père, on peut procéder de la même manière sur la couche précédente. Prenons par exemple l'étape de l'algoritme où nous arrivons à la couche 2. Le vecteur d'utilité pour le premier état de cette couche a été calculé précédemment. Nous allons redescendre dans l'arbre pour le deuxième élément de cette couche et procéder comme expliqué précédemment.

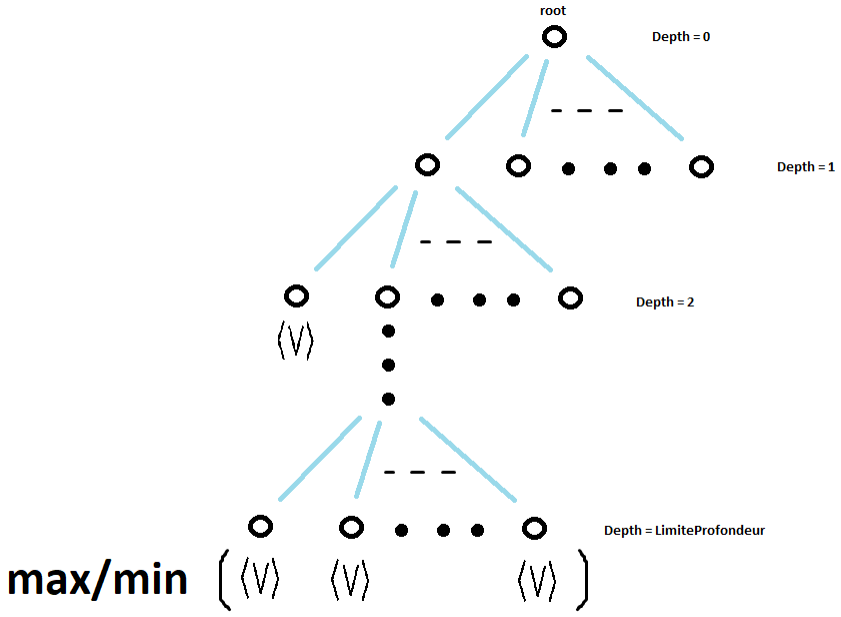


Figure 23: Descente dans l'arbre pour calculer le deuxième élément de la couche 2

[10] Article sur l'algorithme de Paranoïde

- De manière récursive, l'algorithme calcule les vecteurs d'utilité de la couche 2.

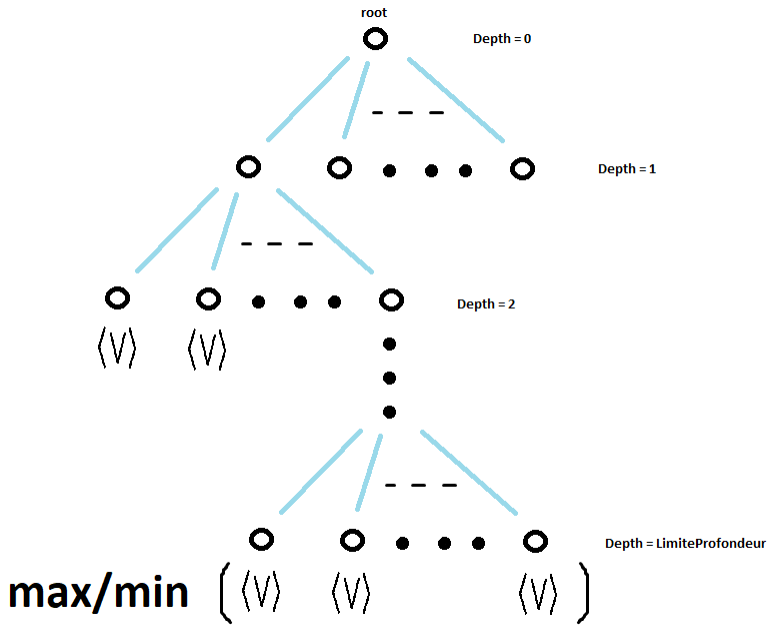


Figure 24: Calcul des vecteurs d'utilité de la couche 2 de manière récursive

- L'algorithme procédera de la même manière pour la couche 1.

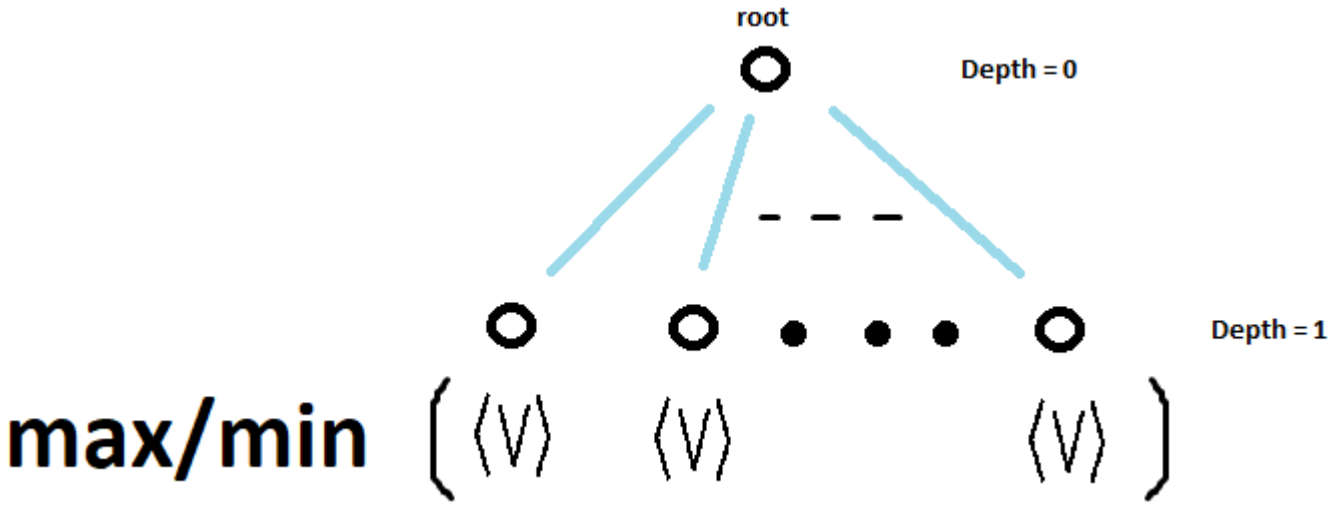


Figure 25: Utilisation de l'algorithme de Paranoïde sur la couche 1

- Nous avons accès à la couche 0 du meilleur vecteur qui permettra à notre IA de faire un choix parmi les actions possibles. Ayant créé une référence entre l'action effectuée et le vecteur calculé, nous pouvons aisément retrouver l'action qui amène à l'état qui nous intéresse.

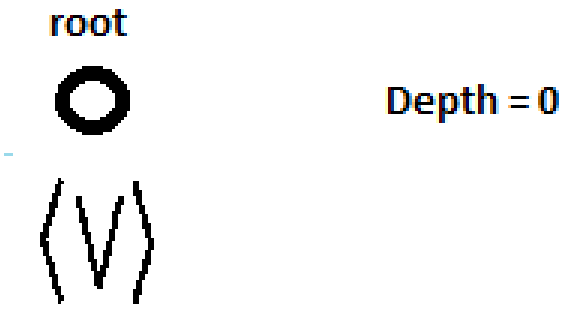


Figure 26: Choix du meilleur état par la couche 0

La corps de notre algorithme se compose des règles *strat/5*, *best_ branch/7* et *worst_ branch/7* qui s'appellent de manière récursive. De plus, ces règles en utilisent d'autres vitales pour leur fonctionnement telles que *best_ vector/3* qui calcule le meilleur vecteur parmi une liste de vecteurs et *worst_ vector/3* qui calcule le pire vecteur parmi une liste de vecteurs. D'autres règles auxiliaires sont importantes aussi à citer telles que *better_ vector/4* qui calcule le meilleur vecteur parmi deux et *worse_ vector/4* qui calcule le pire vecteur parmi 2.

Pour pouvoir appliquer un algorithme tel que le Paranoïde, nous avons dû définir ce qu'est un meilleur vecteur par rapport à un autre. Nous avons fait cela en définissant les heuristiques suivantes :

- Si la valeur d'utilité pour notre joueur est plus grande que la moyenne des valeurs d'utilité du vecteur pour au moins un des deux vecteurs, on choisit le vecteur qui a la plus grande valeur d'utilité pour notre joueur.[11]
- Sinon,
  - Si la différence entre l'utilité du joueur adverse le plus avancé et la moyenne des utilités des joueurs adverses est plus grande que la moyenne des différences des vecteurs d'utilité des joueurs adverses et la moyenne des utilités des joueurs adverses, alors on choisit le vecteur qui pénalise au plus le joueur le plus avancé. [12]
  - Sinon, on choisit le vecteur qui minimise la moyenne des utilités des joueurs adverses. [13]

Ces heuristiques définies ci-dessus nous permettent d'essayer d'accroître notre avance si on est en avance ; de limiter la progression du joueur le plus avancé si un adversaire fait une échappée et de ralentir au plus les adversaires si on est en retard.

[11] Implémenté par *greater_ than_ average/4*
[12] Implémenté par *greater_ than_ avg_ diff_ opp/9*
[13] Implémenté par *lower_ than_ avg_ diff_ opp/9*

4 Conclusions et améliorations possibles

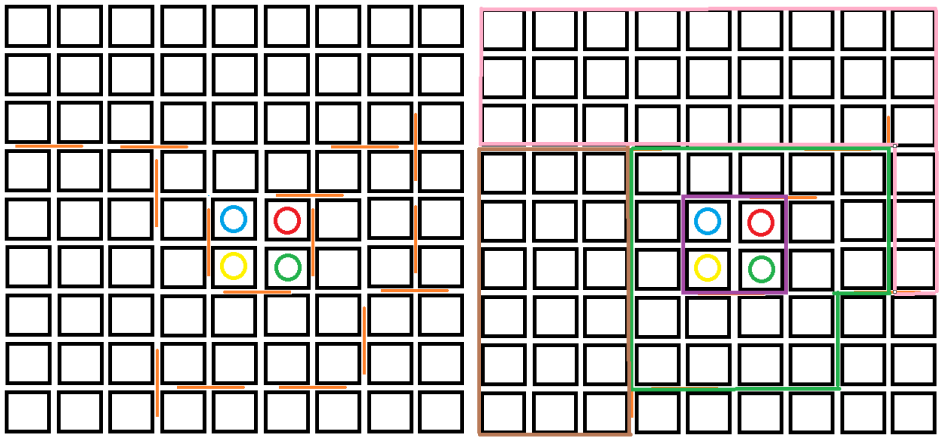
4.1 Quoridor AI

Tout au long du projet, nous avons perfectionné notre maîtrise de la programmation logique avec prolog. De plus, l'implémentation d'une IA pour le jeu Quoridor nous a permis de comprendre et de mettre en pratique les concepts du cours et en particulier ceux liés aux chapitres *"Recherche et Jeux"* et *"Méthodes de recherche"*. Dans un premier temps, nous avons éprouvé de la difficulté à comprendre la manière dont nous pouvions implémenter une IA pour ce jeu. C'est en reparcourant le cours, en comprenant scrupuleusement ses concepts et en procédant étape par étape tout en ayant l'objectif en vue que nous sommes parvenus à obtenir ce résultat. Cette intelligence artificielle n'est pas parfaite mais nous avons compris et savons implémenter tous les concepts nécessaires à son amélioration (bien entendu ceux appris au cours).

Une première amélioration qui peut y être apportée se situe au niveau de l'algorithme minimax pour permettre un élagage alpha-beta. Ceci permettrait d'accroître la vitesse de prise de décision de l'IA ou de parcourir un plus grand nombre de couches et obtenir un résultat plus satisfaisant. C'est un point qui était initialement prévu dans notre plan de conception de l'IA mais que, par manque de temps, nous n'avons pas implémenté.

Une seconde amélioration qui peut y être apportée se situe niveau de la fonction d'utilité. En effet, cette fonction rapporte une valeur en fonction d'une équation que nous avons définie. Cette équation peut être affinée et améliorée. De plus, nous sommes partis de 3 heuristiques mais nous en avions initialement prévu d'autres qui sont plus difficiles à implémenter. Par exemple, le fait d'obtenir le tempo [14] sur les autres joueurs peut s'avérer être un grand avantage. Pour continuer, les degrés d'importance que l'on accorde aux heuristiques (C1,C2,C3) peuvent être affinés pour jouer de meilleurs coups. Dans notre cas, nous les avons modifiés quelques fois jusqu'à obtenir un résultat acceptable.

Une troisième amélioration qui peut y être apportée se situe au niveau des utilisations que l'on peut faire de *shortest_path/9*. En effet, avec la manière dont on l'a implémenté, on peut très bien, et très rapidement calculer le plus court chemin entre deux points en passant par des points intermédiaires. Cela pourrait être utile pour diviser le terrain en zones en le couplant à *number_path/7*. Ainsi, notre stratégie pourrait se baser sur l'emplacement des barrières sur le terrain et la détection de nos chemins qui risquent de se faire couper.



(a) État du jeu (b) Zones déterminées à partir de l'état

Nous avons consacré beaucoup de temps à cette partie du projet. Nous sommes fiers de ce que nous en avons produit. Nous avons veillé à tester à chaque étape les nouveaux codes que nous introduisions pour minimiser le temps passé au débuggage bien que ce ne fut pas suffisant pour ne pas passer plusieurs heures voire plusieurs jours dessus dans les étapes les plus complexes. En conclusion, si l'IA était à refaire, nous procéderions avec la même

[14] Utiliser moins de coups pour bloquer les adversaires qu'eux en utiliseraient pour s'en défaire.

stratégie et irions sans doute un peu plus loin mais surtout nous produirions un code plus propre étant donné la maîtrise que nous avons acquise avec la programmation logique.