Quarto!

Elie Duboux
Yann Trividic

Cours Intelligence Artificielle — 2021
Licence 3 Informatique

1. Présentation du jeu

a. Quarto! et ses règles

Quarto! est un jeu de plateau à deux joueurs et se jouant au tour par tour. Ce jeu a été créé par Blaise Muller et publié par Gigamic[1].

Pour gagner au *Quarto!*, l'objectif est d'aligner quatre pièces ayant au moins un point commun entre elles. La principale singularité du jeu est que ce n'est pas le joueur qui choisit la pièce qu'il va placer : c'est son adversaire qui le fait pour lui.

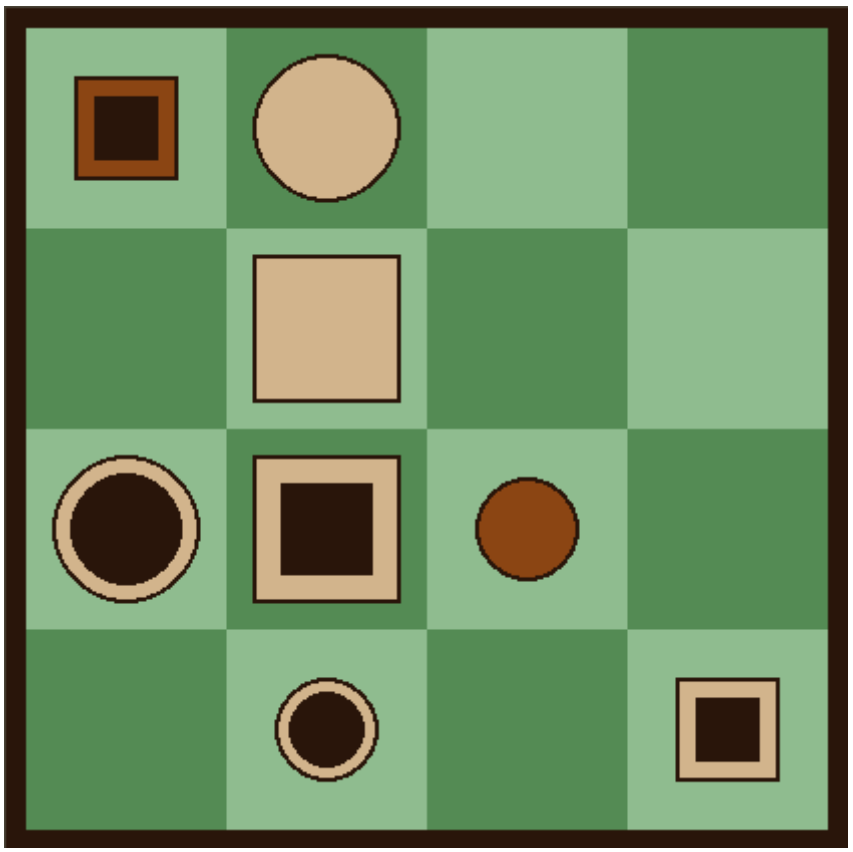


FIgure 1 : Exemple de victoire par la couleur sur la seconde colonne. Le joueur ayant posé la pièce complétant la seconde colonne gagne la partie

Le plateau est constitué de seize cases : quatre lignes et quatre colonnes. Les seize pièces du jeu, toutes différentes, possèdent chacune quatre caractères distincts : large ou étroite, ronde ou carrée, claire ou foncée, pleine ou creuse. Chaque tour se déroule de la manière suivante : le premier joueur choisit une pièce et la donne à son adversaire, qui doit la jouer sur une case libre. Le tour d'après, les rôles s'inversent. Une fois que quatre pièces portant une caractéristique en commun sont alignées (sur une ligne, une diagonale ou une colonne), le joueur qui a posé la dernière pièce gagne.

[1] https://fr.wikipedia.org/wiki/Quarto

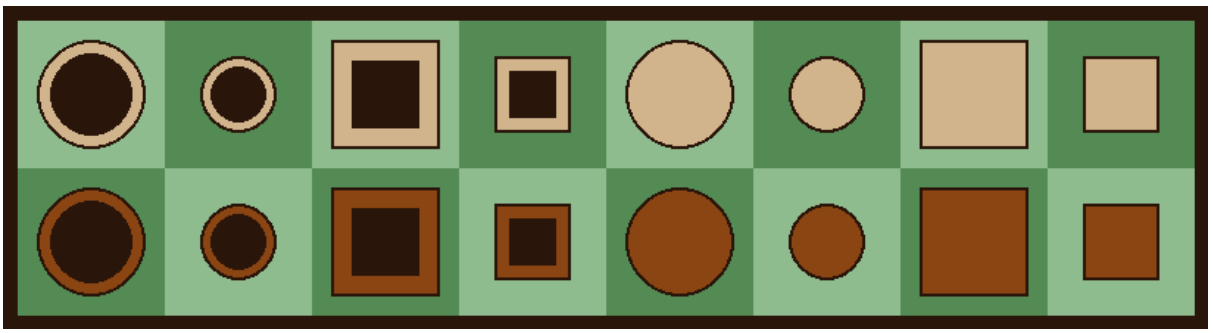


Figure 2 : Les seize pièces à jouer

Il est possible dans certains cas d'arriver à une égalité : lorsque toutes les pièces ont été posées mais qu'aucun des alignements formés ne permet de victoire. En voici un exemple ci-dessous.

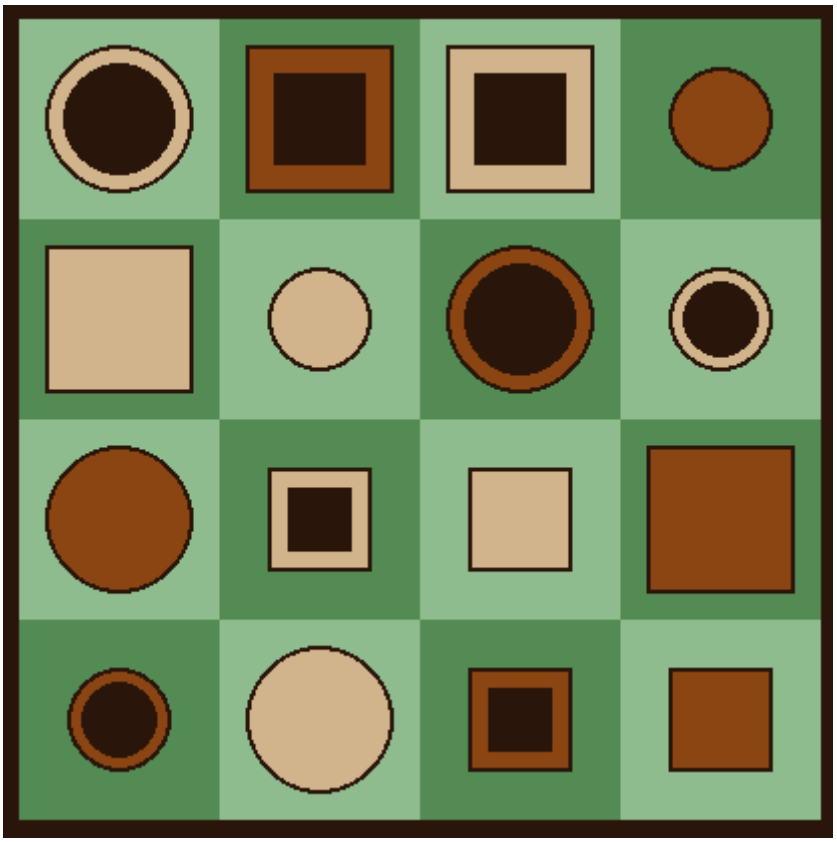


Figure 3 : Exemple de cas de figure où la partie s'est terminée sans vainqueur

Il existe de nombreuses variantes des règles du *Quarto!*, par exemple avec d'autres motifs permettant la victoire (quatre pièces formant un carré, etc.). Nous ne nous attarderons pas dessus dans le cadre de ce projet et nous focaliserons sur la version classique des règles.

b. En termes d'intelligence artificielle

A contrario de certains jeux de plateau, *Quarto!* n'a pas beaucoup fait parler de lui dans le monde scientifique. Quelques articles existent à son sujet, mais rien de comparable à l'intérêt développé pour ses semblables comme les échecs, les dames, le go, etc. Un chercheur en particulier, Luc Goossens, a pris le temps de travailler sur le jeu et l'a résolu : deux joueurs parfaits termineront toujours leur partie sur un match nul (Holshouser et Reiter, 2003).

Ce jeu est d'autant plus intéressant de par l'ordre de grandeur des combinaisons possibles qu'il génère. En effet, étant donné les règles du jeu, et que les seize pièces peuvent être posées sur seize cases différentes sans remise, on arrive, pour les combinaisons à l'ordre de grandeur suivant : $16!^2$, soit environ $4{,}4 \cdot 10^{26}$ combinaisons (les parties se terminant avant d'avoir posé les 16 pièces ne changent pas l'ordre de grandeur). Il est ainsi impossible de parcourir l'espace d'état du jeu dans son entièreté dans un temps acceptable pour un humain, même avec des machines très puissantes.

Une autre métrique est intéressante pour penser le *Quarto!* : son facteur de branchement. Etant donné que les joueurs ont d'abord le choix entre 16 pièces et 16 cases, puis 15 pièces et 15 cases, on arrive à calculer le facteur de branchement moyen en sommant les nombres de 1 à 16 puis en divisant cette somme par 16, soit 8,5. En comparaison, les échecs ont un facteur de branchement d'environ 35 tandis que les dames ont un facteur de branchement moyen de 3 (Plaat et al., 2014), ce facteur de branchement moyen intermédiaire permet l'utilisation de l'algorithme minimax et de l'élagage alpha-bêta. Il est à noter que même si le facteur de branchement moyen est abordable, il est de 15 dans les trois premiers tours, et de 2 dans les deux derniers tours. Cette grande variation implique de traiter différemment le début de partie de la fin de partie, mais nous y reviendrons dans la section *4.c. Minimax*.

2. PEAS

a. Mesure de performance

A l'origine, l'objectif était d'abord de développer un agent focalisé sur l'utilité, maximisant une fonction d'utilité, ou d'heuristique. Dans ce cas-là, notre agent aurait essayé de construire une suite d'actions afin de maximiser l'utilité obtenue (et ainsi obtenir la victoire).

Cependant, nous avons rencontré des écueils lors du développement de la fonction d'heuristique, et l'agent proposé ici est un agent focalisé sur l'objectif. Son unique objectif est de gagner la partie en construisant une suite d'actions lui permettant d'atteindre son objectif. L'objectif étant la victoire, la seule mesure de performance est donc la victoire (le nombre de coups pour obtenir la victoire n'ayant pas d'importance).

b. Environnement

L'environnement défini par le *Quarto!* est un environnement :

- **Totalement observable :** seuls sont utilisables le plateau de jeu et les 16 pièces.
- **Multi agent :** ce jeu se joue à deux, un agent contre un autre agent.
- **Déterministe :** il n'y a pas de place pour le hasard dans le *Quarto!*, chaque choix effectué par un agent obtient le résultat escompté sans aucune variation possible.
- **Séquentiel :** chaque état découle du coup décidé précédemment, et une suite séquentielle de coups (ou d'états) aboutit sur la fin de la partie.
- **Statique :** l'environnement est statique, il n'évolue pas sans action des agents.
- **Discret :** il existe un nombre fini d'états, et aucun entre-deux entre ceux-ci.

c. Actionneurs

Un agent dispose de deux leviers pour agir sur son monde. Lorsque c'est le moment de poser une pièce choisie par l'adversaire sur le plateau, l'agent peut poser cette pièce dans une des cases libres du plateau. Ensuite, lorsque c'est le moment de choisir une pièce pour l'adversaire, il peut choisir une pièce parmi les pièces encore disponibles.

d. Capteurs et structure des états

Chaque agent dispose de trois capteurs :

- La représentation logique de la grille 4x4 avec les positions des seize pièces.
- La représentation logique des pièces encore non posées.
- La représentation logique de la pièce sélectionnée par l'adversaire s'il y en a une.

Informatiquement parlant, nous avons fait le choix de représenter logiquement une pièce par quatre valeurs binaires : une pour la largeur, une autre pour la rondeur, une pour la couleur et finalement une pour le remplissage.

La structure des états est représentée par un tuple nommé `game_state` comprenant trois éléments : le premier est un objet `Board` (voir la documentation du code), la `game_board`, qui contient les positions des différentes pièces déjà posées sur le plateau du jeu. Le second élément est une autre instance de l'objet `Board`, la `storage_board`, contenant les pièces encore disponibles. Le dernier élément du tuple correspond aux coordonnées en abscisse et en ordonnée de la pièce sélectionnée par l'adversaire dans la `storage_board`.

3. Programme

Le programme sur lequel s'adosse le projet a été développé en Python. Nous avons choisi Python pour sa rapidité de développement, sa communauté très active en intelligence artificielle et la bibliothèque pygame permettant de développer rapidement des jeux vidéo.

Le code est disponible sur GitHub à cette adresse (github.com) sous licence MIT[2], un fichier README est fourni pour vous accompagner dans l'installation du programme. Vous pourrez aussi y trouver la documentation du programme au format HMTL.

a. Utilisation

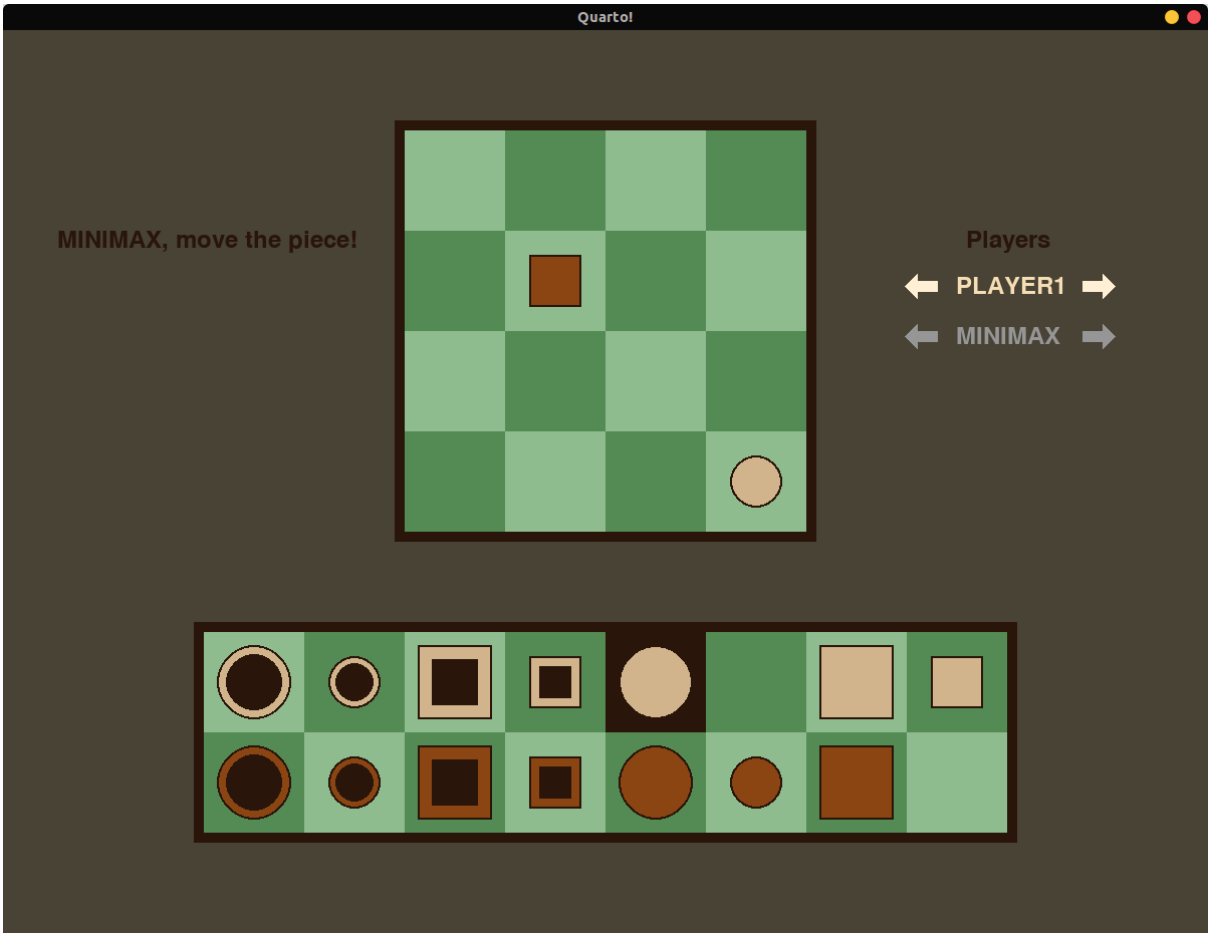


Figure 4 : Vue du programme

Le programme se compose d'une unique fenêtre. Celle-ci permet de jouer autant de parties que l'utilisateur le souhaite.

Le plateau de jeu est situé dans la partie centrale de la fenêtre ; c'est là que sont posées les pièces sélectionnées. La partie inférieure de la fenêtre permet de sélectionner des pièces.

[2] https://fr.wikipedia.org/wiki/Licence_MIT

Ici, on peut voir que le joueur PLAYER1 (humain) a sélectionné une pièce : elle est identifiée par la case marron. C'est donc au tour du joueur MINIMAX (agent logique) de jouer. Ce joueur étant un agent logique, il suffit au joueur humain d'attendre que les calculs aient été effectués pour qu'il puisse jouer de nouveau. Les instructions données aux différents joueurs sont actualisées dans la partie gauche de la fenêtre.

Dans la partie droite de la fenêtre, il est rendu possible à l'utilisateur de changer d'adversaire : il n'est possible de changer le d'adversaire seulement lorsque ce n'est pas le tour dudit adversaire. Il existe quatre types d'adversaires possibles : les humains (PLAYER1 et PLAYER2), et les trois types d'agents décrits dans la section *Algorithmes*, à savoir RANDOM, NOVICE et MINIMAX.

b. Diagramme de classe

L'architecture du programme suit le diagramme de classes ci-dessous.

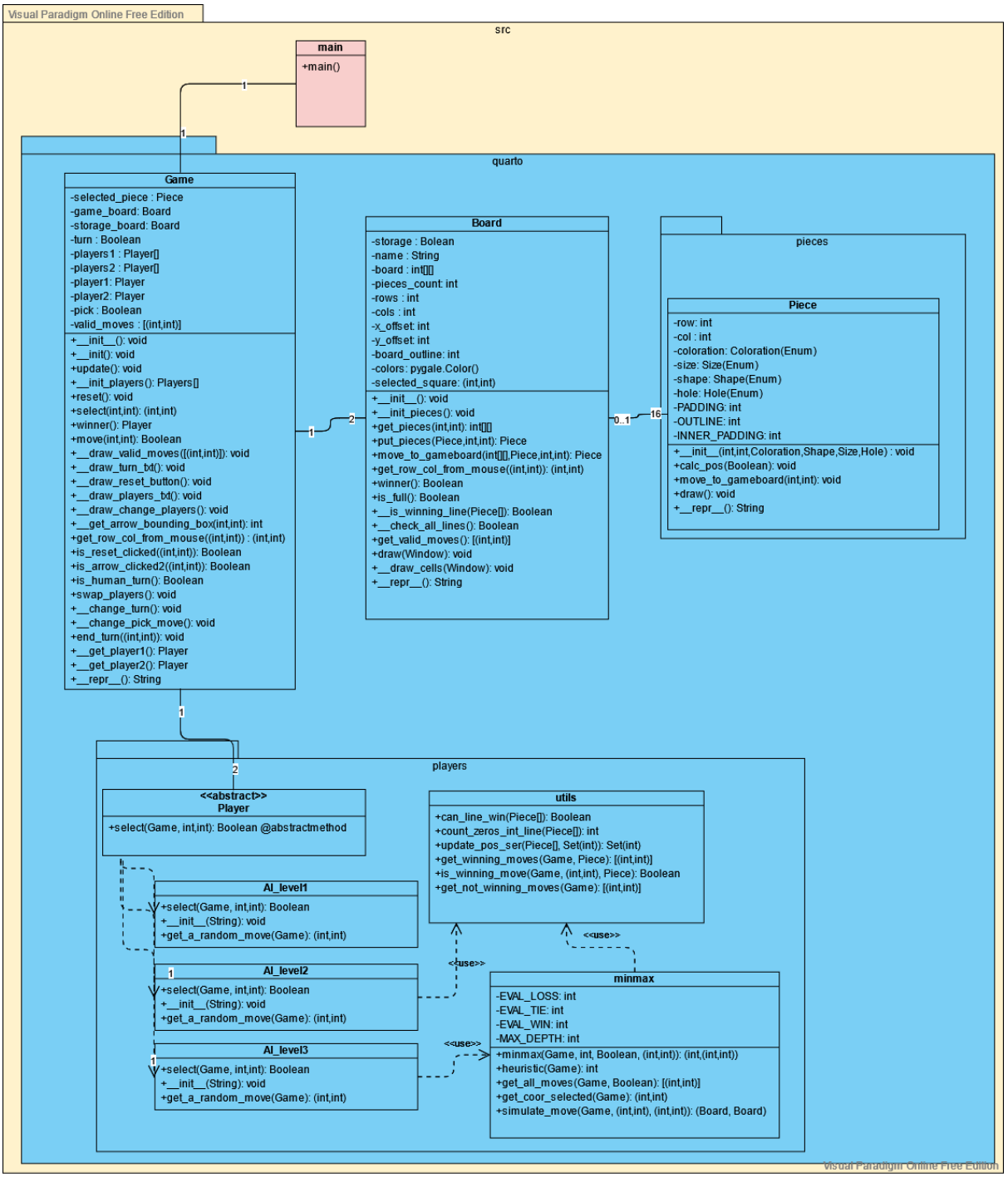


Figure 5 : Diagramme de classes du programme

4. Algorithmes

a. Aléatoire (niveau 1)

Cet agent n'est pas à proprement parler une intelligence artificielle. Nous qualifions de niveau 1 ce joueur n'effectuant que des coups aléatoires appartenant à la liste des coups possibles. Ne nécessitant aucun autre calcul, il s'agit donc d'un joueur extrêmement basique qui n'a pas été doté de quelconque capacité d'analyse.

Quiconque ayant pris connaissance des règles du jeu peut battre cet agent.

b. Novice (niveau 2)

Cette intelligence artificielle de niveau 2 est capable de prendre une décision si cette dernière peut mettre fin à la partie. C'est-à-dire que cet agent ne joue que des coups aléatoires sauf dans deux cas précis :

- **Phase de pose :** L'adversaire a donné à l'agent une pièce permettant de gagner et l'agent met fin à la partie en posant la pièce dans l'une des cases offrant la victoire.
- **Phase de sélection :** L'adversaire pourrait gagner si l'agent choisissait certaine(s) pièce(s). Dans ce cas, l'agent choisit l'une des pièces ne donnant pas la victoire à l'adversaire. Cette pièce est choisie aléatoirement.

Il s'agit donc d'un agent capable d'un minimum de raisonnement et ne donnant pas de victoire facile. Elle possède un niveau correct mais est facilement battable par un joueur un peu aguerri.

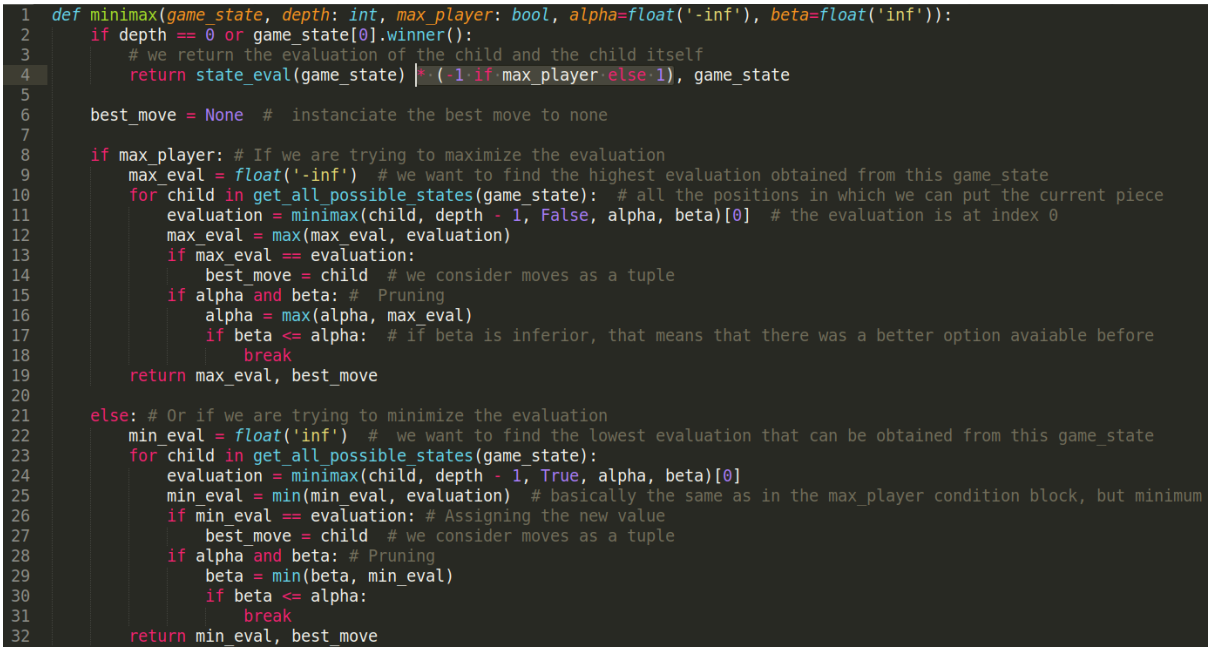
c. Minimax (niveau 3)

L'agent de niveau 3 commence à être plus intéressant algorithmiquement. Son algorithme repose sur une adaptation au *Quarto!* de l'algorithme minimax[3] avec en addition l'élagage alpha-bêta.

i. Minimax et élagage alpha-bêta

L'algorithme utilisé ici est assez classique et suit presque complètement la version proposée dans le cours. Il a cependant recours à une variation pour pallier le fait que les joueurs partagent les pièces du plateau. Voir la section *4.c.ii. Heuristiques* pour plus de détails. Voici ci-dessous l'algorithme utilisé dans le programme (en version simplifiée pour des questions de lisibilité) :

[3] https://fr.wikipedia.org/wiki/Algorithme_minimax



```python
def minimax(game_state, depth: int, max_player: bool, alpha=float('-inf'), beta=float('inf')):
    if depth == 0 or game_state[0].winner():
        # we return the evaluation of the child and the child itself
        return state_eval(game_state) * (-1 if max_player else 1), game_state

    best_move = None  #  instanciate the best move to none

    if max_player: # If we are trying to maximize the evaluation
        max_eval = float('-inf')  # we want to find the highest evaluation obtained from this game_state
        for child in get_all_possible_states(game_state):  # all the positions in which we can put the current piece
            evaluation = minimax(child, depth - 1, False, alpha, beta)[0]  # the evaluation is at index 0
            max_eval = max(max_eval, evaluation)
            if max_eval == evaluation:
                best_move = child  # we consider moves as a tuple
            if alpha and beta: #  Pruning
                alpha = max(alpha, max_eval)
                if beta <= alpha:  # if beta is inferior, that means that there was a better option avaiable before
                    break
        return max_eval, best_move

    else: # Or if we are trying to minimize the evaluation
        min_eval = float('inf')  #  we want to find the lowest evaluation that can be obtained from this game_state
        for child in get_all_possible_states(game_state):
            evaluation = minimax(child, depth - 1, True, alpha, beta)[0]
            min_eval = min(min_eval, evaluation)  # basically the same as in the max_player condition block, but minimum
            if min_eval == evaluation: # Assigning the new value
                best_move = child  # we consider moves as a tuple
            if alpha and beta: # Pruning
                beta = min(beta, min_eval)
                if beta <= alpha:
                    break
        return min_eval, best_move
```

Figure 6 : Adaptation de l'algorithme minimax

Quelques points nécessitent des éclaircissements pour comprendre au mieux ce bout de code.

Le paramètre `game_state` est expliqué en détails dans la section *2.d. Capteurs et structure des états*. Le paramètre `max_player` correspond à un booléen représentant s'il s'agit d'une profondeur où il faut maximiser la valeur de l'évaluation, ou au contraire, s'il faut la minimiser. `alpha` et `beta` sont deux paramètres initialisés respectivement à +∞ et à -∞ `servant à l'élagage`.

L'algorithme est ainsi fidèle à la version présentée dans le cours si ce n'est concernant la ligne 4 et plus particulièrement sa partie soulignée : en effet, étant donné que la fonction d'évaluation ne donne des informations que sur la proximité d'une victoire ou la présence d'un vainqueur, il était nécessaire de discriminer la valeur retournée en fonction du joueur ayant posé la dernière pièce. Ainsi, si `max_player` a pour valeur `False` et que l'état est une victoire (état terminal) alors le joueur cherchant à minimiser l'évaluation des états a perdu et la valeur retournée est `9`. Si `max_player` est à `False` tandis que l'état est une victoire, alors le joueur ayant effectué le premier appel de minimax a perdu la partie dans cet état particulier du jeu ; la valeur retournée est `-9`. Pour plus d'informations, voir la section *4.c.iii. Fonction d'évaluation*.

Comme évoqué précédemment, l'agent développé ici est un agent focalisé sur l'objectif et ne disposant pas de fonction d'heuristique. Cela implique que les états non terminaux ont tous une évaluation à 0, et seuls les états correspondant à une victoire ont sont évalués à une valeur non nulle. L'élagage alpha-bêta permet de gagner un temps de calcul optimal lorsque les états à explorer sont rangés par potentiels décroissants. Dans le cas du *Quarto!*, effectuer ce classement est assez périlleux, et devient même impossible en l'absence d'une

fonction d'heuristique. Ainsi, malgré le gain de temps offert par l'élagage, certains états demandent quand même l'exploration de nombreux états pour parvenir à un choix avisé.

Pour réduire le temps de calcul, nous avons mis en place un système de profondeur adaptative. Ce système permet de faire augmenter la profondeur de recherche plus la partie avance. Au tout premier tour, la profondeur est ainsi de 1 (le premier coup n'a pas d'importance et n'importe quelle pièce peut être choisie pour une efficacité égale), puis la profondeur augmente jusqu'à atteindre une valeur de 4 en fin de partie. Par cette astuce, le temps de calcul a pu être limité pour rentrer dans des limites acceptables dans le cadre d'un jeu de réflexion.

ii. Heuristiques

1. Mohrmann et al. (2013)

Pour ce projet, nous avons dans un premier temps tenté d'utiliser la fonction heuristique décrite par Mohrmann et al. (2013). Cependant, son implémentation posait problème dans l'application de l'algorithme minimax, et la supprimer permettait de régler le problème. La cause du problème rencontré reste encore à être éclaircie et une heuristique a encore à être trouvée pour convenir à nos besoins.

Malgré cela, voici la réflexion qui motivait cette heuristique. Chaque ligne (colonne, ligne diagonale) comportant trois pièces et une case vide représente un potentiel de victoire (ou de défaite). Pour chacune de ces lignes, il s'agit de vérifier si l'une des pièces disponibles pourrait résulter sur un état terminal. Si tel est le cas, alors on ajoute 1 à la valeur retournée (initialisée à 0). Ainsi, au début de la partie, l'heuristique de l'état initial est forcément égale à 0. Ce n'est qu'au moment de poser la troisième pièce que l'heuristique de l'état pourrait être égale à 1 (si les trois premières pièces posées comportent une caractéristique commune ou plus, alors l'heuristique est égale à 1). En suivant ce raisonnement, la valeur maximale générée ne peut être que 7, comme illustré ci-dessous.

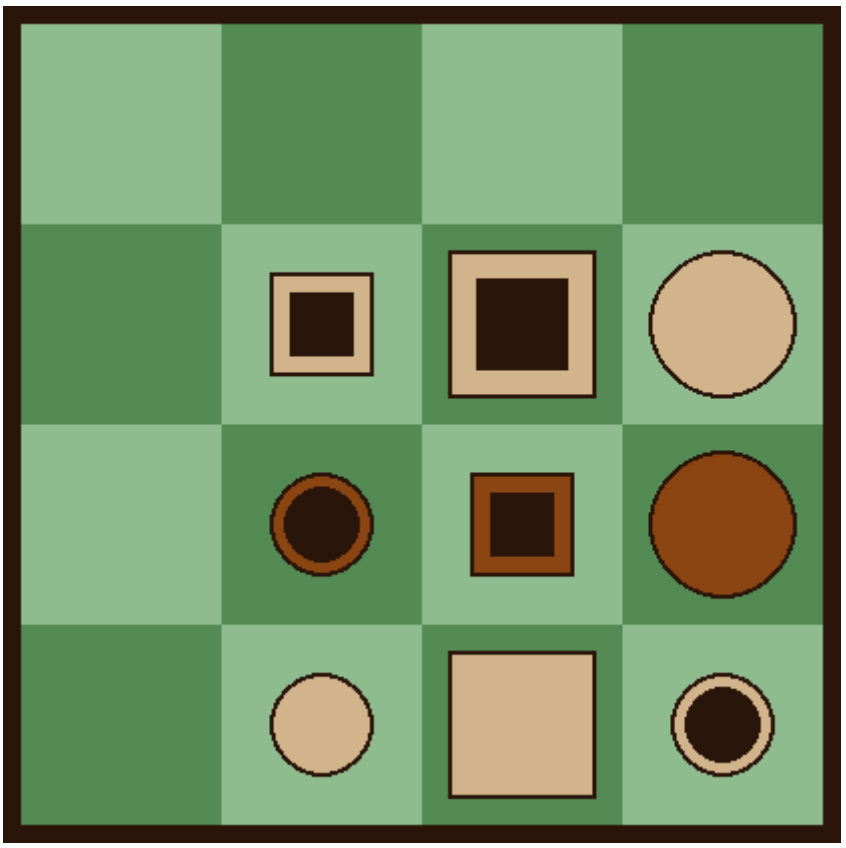


Figure 7 : Etat illustrant un cas de figure où la fonction d'heuristique retournerait la valeur maximale 7 (chaque cellule encore disponible peut résulter sur une victoire).

2. Fressinaud (2013)

L'une de nos référence lors de ce projet, Marien Fressinaud, a eu une approche différente de la nôtre quant à l'heuristique utilisée. En effet, à chaque tour, une évaluation est donnée à chaque case du plateau de jeu en fonction de la pièce donnée par l'adversaire. Une autre évaluation est donnée à chaque pièce disponible quand vient le moment de choisir une pièce.

Ensuite, l'agent pose la pièce avec la plus grande évaluation et donne à l'adversaire celle avec la plus petite. Pour connaître la meilleure position possible pour une pièce, il étudie chacune de ses propriétés, voir section *1.a. Quarto! et ses règles.* Pour maximiser l'évaluation d'une propriété, Fressinaud regarde la position où une ligne est gagnante le plus rapidement. Ensuite il compare ces propriétés avec la pièce donnée par l'adversaire ou celles que l'agent doit donner à l'adversaire.

Quant aux valeurs prises par l'évaluation, 500 correspond à la victoire, 490 à l'égalité, 0 à la défaite, puis de 0 à 480 pour les autres cas possible.

3. Nos réflexions

L'heuristique a été sujette à de nombreuses réflexions. Mais par soucis de temps ou de difficulté d'implémentation, nos différentes idées ont été écartées pour se focaliser sur le cœur de l'algorithme minimax.

Parmi les pistes de réflexions :

- **Une heuristique allant de 0 à 6 :** Cette dernière aurait déterminé pour chaque pièce la case maximisant son score. Le critère est basé sur le nombre de cases qu'il reste à compléter pour gagner sur une ligne. La valeur retournée correspond à : 0 mouvement perdant, 5 à une égalité, 6 à une victoire et de 1 à 4 au nombre de cases restant à compléter suite au le meilleur coup.
- **Une heuristique déterminant le joueur dominant.** Ici, on analyse pour chaque coup, quel joueur domine la partie suite à un certain coup. On regarde quel joueur domine en déterminant quel joueur des deux est celui pouvant compléter au plus vite l'une des lignes.

iii. Fonction d'évaluation

```
36  def state_eval(game_state):
37      if game_state[0].winner():
38          return EVAL_WIN
39      elif game_state[0].is_full():
40          return EVAL_TIE
41      else:
42          return 0
43          # return heuristic(game_state) + 1  #  the + 1 is here to let only the TIE be of 0 value
```

Figure 8 : Fonction d'évaluation

La fonction d'évaluation nommée ici `state_eval` est relativement directe. Elle prend en entrée un argument, `game_state`, qui est une représentation de l'état courant, voir la section *2.d. Capteurs et structure des états.* La première étape va être de vérifier si l'état est terminal, c'est-à-dire s'il y a un gagnant ou une égalité : la méthode `is_full` indique s'il est possible de poser des pièces ou non. Une victoire est évaluée à 9, et une égalité à `0`.

Originellement, si l'état n'était pas terminal, alors la valeur de l'évaluation allait être celle retournée par l'heuristique, à laquelle on aurait ajouté 1 pour que la valeur de l'égalité ne puisse pas se confondre avec d'autres états non-terminaux.

Ainsi, la valeur retournée par la fonction d'évaluation était censée pouvoir prendre pour valeurs tous les entiers compris entre 0 et 9. Dans le cadre de notre agent focalisé sur l'objectif, elle ne peut en prendre que deux : 0 et 9.

5. Tournoi

Avec ces trois agents et un humain, nous avons pu organiser un tournoi. Chaque couple de participants a ainsi pu s'affronter cinq fois, pour un total de trente parties. La lecture de ce tableau s'effectue en lisant le nombre de défaites en colonne et le nombre de victoires en ligne. Le nombre d'égalités est représenté entre parenthèses.

	Humain (Yann)	**Aléatoire**	**Novice**	**Minimax**
Humain (Yann)		5 (0)	3 (0)	2 (1)
Aléatoire	0 (0)		1 (0)	0 (0)
Novice	2 (0)	4 (0)		2 (0)
Minimax	2 (1)	5 (0)	3 (0)	

Table 1 : Résultats du tournoi

Le classement (avec un résultat un peu surprenant !) donne vainqueurs ex-aequo l'agent Minimax et Yann, l'agent novice arrivant 3^{e} de peu et le joueur aléatoire dernier. On aurait pu s'attendre à ce que l'agent Minimax performe mieux, peut-être que les résultats seraient meilleurs en augmentant la profondeur de l'arbre de recherche. En l'état, les résultats montrent qu'elle peut battre des humains de niveau correct, mais pas à toutes les parties.

6. Bilan

a. Ce que nous avons appris

D'un point de vue de développement pur, le projet nous aura appris à développer un jeu en Python et à nous familiariser avec le motif d'architecture des *event listeners*. Concernant l'aspect intelligence artificielle, le projet nous aura permis de nous familiariser avec l'algorithme minimax et l'élagage alpha-bêta, en nous attaquant à une implémentation concrète de ceux-ci.

b. Difficultés rencontrées

Le problème soulevé dans la section *4.c.ii Heuristique* n'a été compris que très tard dans le développement du projet, ce qui a entraîné des retards pour remplir entièrement tous les objectifs que nous nous étions fixés. Finalement, ce problème a émergé car nous avons mis la charrue avant les bœufs : la fonction heuristique a été développée et intégrée en premier, avant même d'avoir un algorithme minimax opérationnel. La prochaine fois, nous commencerons par la base et améliorerons le projet par incréments.

De manière générale, les réflexions autour des différentes heuristiques à utiliser pour le *Quarto!* ont été laborieuses. En effet, dans la plupart des jeux de plateau, les joueurs ont des pièces qui leurs sont propres et il est souvent assez trivial, pour celui qui connaît les règles de comprendre, qui a l'avantage à partir de la lecture du plateau de jeu. Avec *Quarto!*, ce n'est pas aussi évident car tout est mutualisé. Ce principe, peu habituel, nous aura posé beaucoup de problèmes de réflexions et demandé un temps d'adaptation important.

c. Améliorations possibles

De nombreux points de notre projet sont améliorables. L'agent de niveau 3 (minimax) est très lent à l'exécution, de nombreuses techniques d'optimisation pourraient être utilisées comme l'utilisation d'une heuristique, d'une table de transpositions, de la variation negamax, ainsi que des améliorations sur la structure même du programme en allégeant notre représentation des états.

D'un point de vue logiciel, d'autres points pourraient être développés comme un mode tournoi.

7. Bibliographie

Fressinaud, Marien. 2013. n.d. "Quarto et Intelligence Artificielle / Marien Fressinaud." Accessed April 23, 2021. https://marienfressinaud.fr/quarto-et-intelligence-artificielle.html.

Holshouser, Arthur, and Harold Reiter. 2003. "Quarto without the Twist," November.

Mattiussi, Julien. n.d. "Découverte Du Langage Go Pour Ma Deuxième Semaine d'intégration : Go Go Quarto Ranger !" Accessed April 23, 2021. https://marmelab.com//blog/2018/10/09/go-go-quarto-ranger.html.

Mohrmann, Jochen, Michael Neumann, and David Suendermann. 2018. "An Artificial Intelligence for the Board Game 'Quarto!' in Java." In , 141–46. https://doi.org/10.1145/2500828.2500842.

Plaat, Aske, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. 2014. "Nearly Optimal Minimax Tree Search?" *ArXiv:1404.1518 [Cs]*, April. http://arxiv.org/abs/1404.1518.