Rapport de stage

BUT INFORMATIQUE
3e Année

Parcours RAVCD (A)

Enseignant Référent : Margaret Borthwick
Maître de stage : Kevin Bouchard
Etudiant : Ewan Burasovitch

Tables des matières

I. Remerciements..... 3
II. Le chemin vers la mission..... 3
1. Du Début..... 3
2. l'Université du Québec à Chicoutimi..... 4
III. Context du stage..... 4
IV. L'equipe..... 5
V. Déroulement du stage..... 6
1. Présentation Projet..... 6
2. Les premières recherches..... 8
3. Mise en place..... 11
4. Réalisation..... 12
4.1 - Game..... 13
4.2 - Algorithme..... 14
4.3 - State..... 14
4.4 - Nœud..... 15
4.5 - Arbre de Jeu..... 16
4.6 - Minimax..... 17
4.7 - Monte carlo..... 21
4.8 - Les Test..... 26
4.9 - Outils de visualisation..... 28
5. Les difficultées dans l'adversité..... 30
6. Enseignement en mémoire..... 30
VI. Perspective..... 32
VII. Bilan..... 32
1. Conclusion..... 33

Tables des Figures

Figure 1. Python 9

Figure 2. Github 9

Figure 3. Trello 9

Figure 4. Presentation Trello 10

Figure 5. Présentation Github 10

Figure 6. Partie MIN MAX 12

Figure 7. Diagram UML 13

Figure 8. Noeuds vs Etat 16

Figure 9. Arbre de jeux 17

Figure 10. Propagation des valeurs 19

Figure 11. Démonstration de la profondeur des Échec 20

Figure 12. Enfant et parent dans un arbre 21

Figure 13. Exemple fonctionnement Minimax 22

Figure 14. Les quatres étapes de Monte Carlo 24

Figure 15. Partie de test Minimax 27

Figure 16. État d'un tic tac toe 29

Figure 17. Partie Monte Carlo 29

Figure 18. Outils de visualisation 31

Figure 19. Tâches de la semaine 34

I. Remerciements

Afin de commencer dans les règles, je voudrais remercier M. Kevin Bouchard, professeur d'informatique et de mathématiques, pour avoir encadré ce stage.

Je tiens également à remercier toute mon équipe pour l'aide et le soutien apportés tout au long du projet, ainsi que pour le travail fourni et la collaboration.

Enfin, je souhaite remercier les membres de l'équipe pédagogique du département informatique de l'IUT de Bayonne et du Pays Basque, en particulier ma professeure référente, Mme Borthwick, pour son aide et son encadrement.

II. Le chemin vers la mission

1. Du Début

Je m'appelle Ewan Burasovitch, j'ai 20 ans, et l'informatique a toujours été une passion et un objectif professionnel pour moi.

En seconde, à mon arrivée au lycée, j'ai pris une option liée à l'informatique afin d'affirmer mon choix pour cette voie. Puis, en première, j'ai choisi la filière STI2D afin de prendre la spécialité SIN pour la dernière année de bac.

Une fois mon bac obtenu, je me suis dirigé vers le Bachelor Universitaire de Technologie en informatique à l'IUT de Bayonne et du Pays Basque. Dans cette formation, plusieurs opportunités me sont proposées, dont l'une est un double diplôme au Canada pour la troisième année de BUT. J'intègre donc l'UQAC en troisième année de BUT.

2. l'Université du Québec à Chicoutimi

Fondée en 1969, l'Université du Québec à Chicoutimi (UQAC) est une institution publique francophone faisant partie du réseau de l'Université du Québec. Sa mission principale repose sur trois axes : la formation universitaire, la recherche scientifique et le développement régional.

L'université propose plus de 200 programmes, allant du certificat au doctorat, en passant par les baccalauréats et les maîtrises. Ces formations couvrent un large éventail de domaines, tels que les sciences, les arts, l'éducation, l'ingénierie et les technologies de l'information.

Dans le cadre de ce stage, nous serons intégrés au pôle de recherche en informatique. Ce pôle est principalement animé par François Lemieux, directeur du département d'informatique et professeur, ainsi que par Justine Lévesque, coordonnatrice et agente de stage du département d'informatique et de mathématiques.

III. Context du stage

Dans le cadre de ces études à l'UQAC, nous devons réaliser un stage d'un mois, mis en place par l'UQAC elle-même. Cependant, dans certains cas, il est possible de prolonger ce stage afin de satisfaire les exigences des universités françaises. C'est ainsi que j'ai obtenu mon stage.

Ce stage a donc pour but de mettre en pratique des compétences acquises dans les deux universités, française et canadienne. L'UQAC a choisi un sujet orienté Programmation et Algorithmique. Je vais donc devoir mobiliser et appliquer ces compétences.

Je vais également démontrer les compétences acquises à l'IUT de Bayonne et du Pays-Basque, et mises en pratique lors de mon stage, en particulier :

- Réaliser un développement d'application
- Optimiser des applications
- Collaborer au sein d'une équipe informatique

Pour commencer, nous ferons la présentation des membres de l'équipe ainsi que la mienne. Je vais vous parler du sujet proposé par l'UQAC, puis des méthodes qui ont été mises en place. Puis les différentes étapes du projet ainsi que les enseignements tirés de cette expérience et les perspectives d'avenir. Afin de finir sur un bilan du travail effectué.

IV. L'equipe

Kevin Bouchard est, comme indiqué dans les remerciements, professeur et responsable de mon stage. Il a le rôle de superviseur de notre stage ; c'est également lui qui a proposé le sujet du stage. Le projet est composé de plusieurs rôles, que nous allons présenter avec leurs descriptions respectives. Le rôle de superviseur sera nommé "Boss" durant le projet, et c'est bien sûr le professeur Bouchard qui l'occupera.

Le Boss est souvent exigeant et ne comprend souvent pas très bien ce qui se passe concrètement sur le terrain. Il demande parfois des choses impossibles à faire. Notre équipe doit toujours s'assurer de maintenir le bonheur du Boss à un niveau élevé. Par contre, il faudra parfois apprendre à lui dire non et à justifier nos choix. Si le Boss demande une rencontre spéciale, nous devons être prêts et organisés pour lui répondre à satisfaction.

Nous avons désormais constitué l'équipe, exclusivement composée d'élèves en stage. Ces stagiaires ne suivent pas tous la même filière et n'ont pas les mêmes exigences en termes d'horaires à effectuer pour valider leur stage. Ce que nous allons voir va compliquer les choses, notamment lorsque la moitié des membres ne sont pas issus de filières de programmation.

Benjamin Rifki prend le rôle de l'Orateur. Il est le lien direct entre l'équipe et le Boss. Chaque semaine, il recueille les comptes-rendus de tous les membres et les synthétise pour présenter un portrait fidèle de l'avancement du projet. C'est lui qui défend le travail de l'équipe et met en valeur les progrès réalisés. Il doit être clair, concis et capable de répondre aux questions du Boss sans flancher. Sa présentation hebdomadaire est un moment clé pour maintenir la confiance du Boss envers l'équipe.

Hugo VICENTE est Le Destructeur, il a une mission simple, casser tout ce que l'équipe construit. Il teste, pousse le système dans ses retranchements, et cherche activement les failles. Sa cruauté envers le code est un gage de qualité pour le projet. Chaque semaine, il doit rendre compte des bogues identifiés et alerter l'équipe pour qu'ils soient corrigés. Il est le garant de la robustesse du projet, même si son travail peut faire grincer des dents.

Le rôle du clean coder était effectué par martin queval, Le CleanCoder veille à la propreté et à la cohérence du code produit par l'équipe. Il dirige les revues de code et garantit que chaque morceau de programme respecte les bonnes pratiques. Il doit favoriser un esprit critique sain, encourager les améliorations et arbitrer en cas de désaccord. Son jugement est final sur les décisions techniques prises durant ces séances. Il joue un rôle essentiel pour éviter que le projet ne devienne un amas de scripts chaotiques.

Melissa Djenadi était simple développeuse, elle effectuait les tâches qui lui étaient attribuées et participait aux réunions. Une membre active du projet.

À présent, nous arrivons aux membres non développeurs. Ces membres sont issus d'une filière sans programmation. Ils seront attribués à des tâches de documentation, tout en essayant de suivre le projet à travers les sessions de peer coding ou les réunions.

- Axel OZANGE
- Anaïs Dariane Nikièma
- Mathys Lohezic

Et pour finir moi même, Ewan Burasovitch, je représente le rôle d'oracle responsable de la planification stratégique du projet. Chaque début de semaine, il discute avec chaque membre pour s'assurer que le travail est bien réparti, cohérent et aligné avec les priorités. Il doit prévenir les doublons, valider la pertinence des tâches et anticiper les obstacles. Son objectif est d'assurer que le temps de chacun est utilisé efficacement. C'est également lui qui transmet le plan hebdomadaire au Boss dans les temps.

V. Déroulement du stage

1. Présentation Projet

À l'UQAC, les stages constituent un levier important pour faire progresser les recherches des professeurs. Ainsi, chaque projet est étroitement lié à l'encadrement et à la direction du professeur responsable. Monsieur Kevin Bouchard nous a réunis le vendredi 10 janvier 2025 pour nous présenter les grandes lignes du projet. Ce dernier s'inscrit dans le cadre des travaux de recherche en algorithmique qu'il mène au sein de l'université.

Le projet PyAdverseSearch a pour objectif de concevoir une librairie Python, un outil de développement réutilisable que d'autres programmeurs peuvent intégrer dans leurs projets. Celle-ci spécialisée dans les algorithmes d'exploration adverse, exclusivement ceux utilisés dans des contextes de jeux à information incomplète. Cette librairie se veut à la fois modulaire et extensible, afin de permettre aux utilisateurs d'expérimenter différentes stratégies de décision, d'analyser leurs performances, et de faciliter la mise en œuvre de nouveaux algorithmes. Le projet comprend le développement d'outils de simulation, de visualisation et d'analyse, avec un accent particulier sur la qualité du code, la documentation, ainsi que la rigueur des tests.

L'enjeu principal de ce projet réside dans la création d'une solution technique répondant aux besoins de la communauté scientifique et éducative. En effet, les algorithmes d'exploration adverse sont largement utilisés dans le domaine de l'intelligence artificielle, notamment pour la programmation de jeux comme les échecs, le go, ou encore des jeux plus simples comme le tic-tac-toe. Cependant, il n'existe pas de librairie Python complète et accessible qui permette aux étudiants et chercheurs d'expérimenter facilement avec ces algorithmes.

Notre mission consiste donc à développer cette librairie en respectant les standards de développement Python moderne. Cela inclut la possibilité d'installer la librairie via pip qui est un outil standard pour installer ces librairies, ce qui facilitera grandement son adoption par la communauté. Le projet nécessite également la création d'un dépôt GitHub pour gérer le travail en équipe et potentiellement accueillir des contributions externes.

Pour cela, le sujet du stage aborde plusieurs éléments à réaliser tout au long du projet. Ces jalons serviront de repères pour évaluer notre progression et nous assurer que nous restons alignés avec les objectifs fixés. Chaque élément sera abordé à des moments clés du développement. Les voicis :

1. Étudier comment créer une librairie Python. En outre, un aspect crucial de ce projet sera de garantir que la librairie soit facilement installable via pip, facilitant ainsi son adoption par la communauté Python.

2. Créer un dépôt GitHub afin de gérer le projet en équipe et éventuellement accepter des collaborateurs externes.

3. Noeuds (Nodes) : La librairie fournira une classe pour représenter les différents états ou configurations possibles dans le jeu. Ces noeuds seront interconnectés pour former un arbre de recherche, permettant aux algorithmes d'exploration de naviguer à travers les différentes décisions possibles. La structure suggérée est décrite dans le livre ou dans la présentation PPT Résolution de problèmes par l'exploration – diapo 21.

4. Problèmes : La librairie devra fournir une classe pour définir les problèmes de façon formelle. Les éléments essentiels qui constituent un problème se retrouvent dans le PPT Exploration adverse à la diapo #4. Cependant, garder en tête que certains problèmes pourraient avoir des contraintes particulières (e.g., durée max pour choisir une action).

5. Heuristiques : Pour évaluer la qualité des différents états du jeu, des heuristiques seront fournies ou définissables. Ces fonctions permettront d'estimer la valeur d'un état donné, aidant ainsi les algorithmes à prendre des décisions informées lors de l'exploration de l'arbre de recherche.

6. Structure pour implémenter des algorithmes : La librairie offrira une infrastructure permettant d'implémenter facilement une variété d'algorithmes d'exploration adverse, tels que l'algorithme Minimax, l'élagage alpha-bêta, Monte Carlo Tree Search (MCTS), etc. Cette structure permettra aux utilisateurs d'expérimenter avec différents algorithmes et de les comparer en termes de performance et d'efficacité dans divers scénarios de jeu.

7. Outils : Vous devrez aussi prévoir divers outils et utilitaires pour le développement. Par exemple, la possibilité de voir l'arbre d'exploration, les statistiques sur le déroulement (temps, nombre de nœuds explorés, etc.) ou même des outils pour

8. Exemples : La librairie devrait fournir au moins 1 ou 2 exemples fonctionnels. Je suggère minimalement le jeu du Tic-Tac-Toe (Morpion) et/ou le jeu Puissance 4.

Les technologie que l’on devra utiliser seront donc très simple pour le projet :

Python est un langage de programmation que l’on va utiliser pour sa simplicité et sa polyvalence. Il permet de créer des programmes informatiques dans de nombreux domaines, notamment l'intelligence artificielle et l'analyse de données. Les programmes Python peuvent être partagés sous forme de "librairies", des outils réutilisables que d'autres développeurs peuvent facilement installer et utiliser dans leurs propres projets.

Figure 1. Python

GitHub est une plateforme qui permet de stocker et partager du code informatique en équipe. Nous allons l’utiliser pour la collaboration entre développeurs en gardant un historique des modifications et pour pouvoir travailler simultanément sur le même projet sans conflits.

Figure 2. Github

Trello est un outil de gestion de projet qui utilise des tableaux visuels pour organiser les tâches. C’est grâce à ce logiciel que l’on pourra voir l'avancement du travail avec le déplacement des "cartes" de tâches à travers différentes colonnes comme "À faire", "En cours", "Terminé".

Figure 3. Trello

2. Les premières recherches

Mise en Place du Projet

La première semaine a été consacrée à la collecte des premières informations et ressources nécessaires aux recherches, ainsi qu'à la mise en place des outils de collaboration avec l'équipe. Nous avons établi un groupe de discussion, un tableau Trello pour la gestion des tâches, et un dépôt GitHub pour le versioning du code.

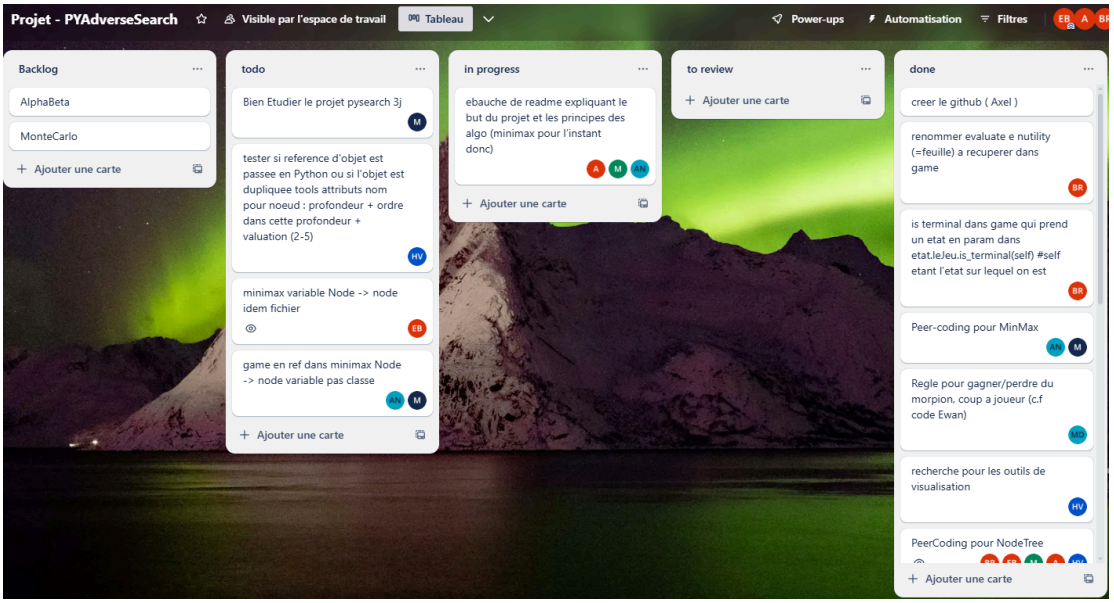


Figure 4. Presentation Trello

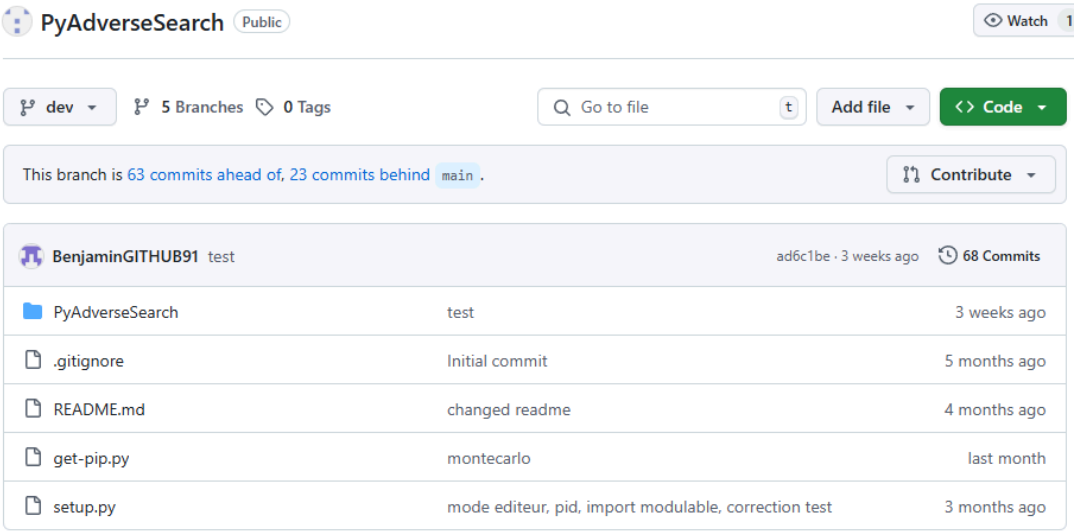


Figure 5. Présentation Github

Monsieur Kevin Bouchard nous a fourni plusieurs ressources documentaires pour débuter nos recherches, notamment :

- **EXPLORATION ADVERSE**
- **EXPLORATION INFORMÉE**
- **RÉSOLUTION DE PROBLÈMES PAR L'EXPLORATION**

Nous avons décidé que le premier algorithme à implémenter serait Minimax, en raison de sa simplicité relative et de sa facilité de compréhension et de mise en œuvre.

Analyse du Document "Exploration Adverse"
Le document "EXPLORATION ADVERSE" correspond exactement à notre domaine d'étude. Ce document a servi de point d’ancrage pour nos premières réflexions, notamment par sa structuration claire des jeux adverses.

Vocabulaire et Concepts Fondamentaux
Un jeu adverse est un type de jeu où deux joueurs s'affrontent en prenant des décisions tour à tour, chacun cherchant à maximiser ses propres gains au détriment de son adversaire. Ces jeux se caractérisent par trois éléments fondamentaux :

- l'alternance des tours entre deux joueurs,
- Des objectifs diamétralement opposés où la victoire de l'un signifie nécessairement la défaite de l'autre,
- Une structure dite "à somme nulle" où tout avantage gagné par un joueur représente une perte équivalente pour son adversaire.

Les exemples les plus familiers de jeux adverses incluent les échecs, les dames, le tic-tac-toe, le puissance 4 ou encore le jeu de go. Dans le contexte informatique, ces jeux présentent un intérêt particulier car ils permettent de développer des algorithmes capables de prédire et d'optimiser les stratégies de jeu, en anticipant les réactions de l'adversaire.

Un jeu adverse se définit d'abord par son état initial, qui correspond à la configuration de départ du plateau ou à la position initiale des différents éléments. La fonction de succession décrit ensuite tous les mouvements légaux possibles à partir de n'importe quel état du jeu. Pour savoir quand une partie se termine, on utilise un test d'état terminal qui n'est pas forcément un test de victoire au sens classique, mais simplement un critère pour identifier les situations où le jeu s'arrête.

La fonction d'utilité joue un rôle central car c'est elle qui attribue une valeur numérique à chaque état final, permettant ainsi d'évaluer si une position est favorable ou non. Tous ces éléments s'organisent dans ce qu'on appelle un arbre de jeu, une structure qui représente graphiquement l'ensemble des scénarios possibles, où chaque nœud correspond à un état et chaque liaison à une action. Les deux joueurs qui s'affrontent seront toujours Min et Max.

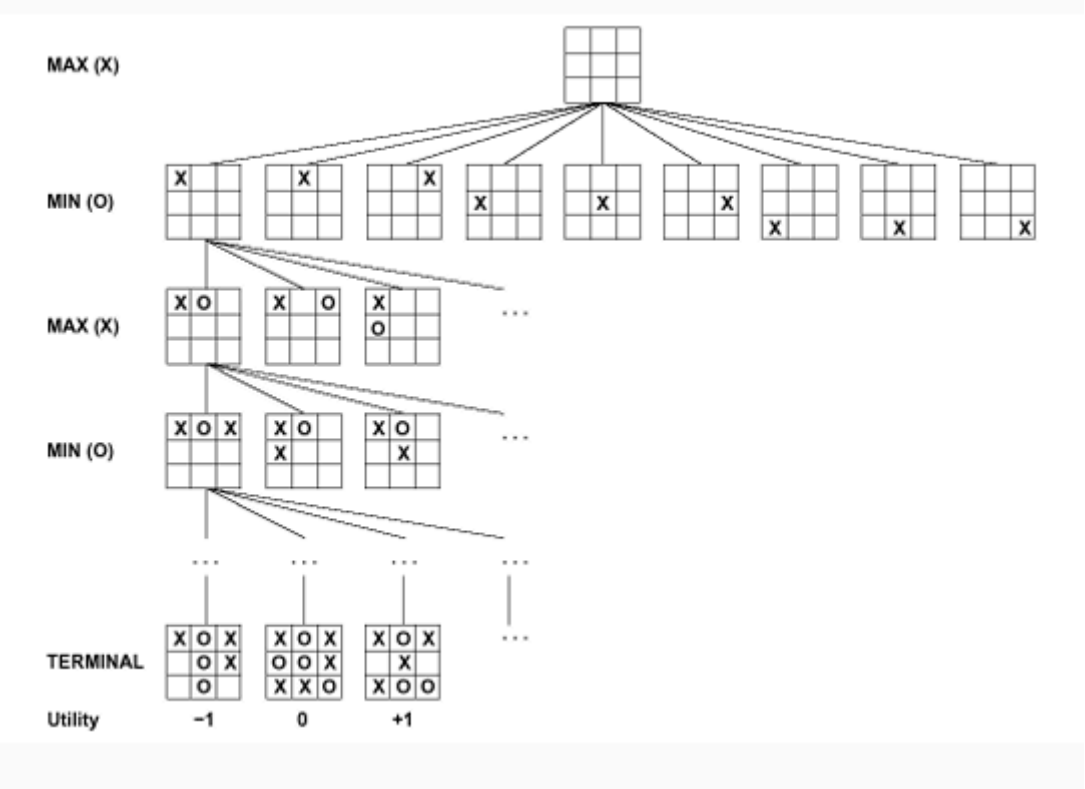


Figure 6. Partie MIN MAX

3. Mise en place

Lors de notre première réunion d'équipe, nous avons mis en commun les informations collectées au cours des deux premières semaines de recherche. Cette séance de travail a permis d'établir une première feuille de route pour l'implémentation de l'algorithme Minimax. L'objectif était de structurer le projet en composants fonctionnels distincts, chacun pris en charge par un membre de l'équipe. C'est ainsi qu'ont émergé les premières classes fondamentales du projet.

- Classe game
- Classe Minimax
- Classe État (State)
- Classe Noeud (Node)
- Classe Arbre (Tree)
- Module Tools
- Fichiers de documentation

D’autres classes vont être ajoutées au fil du projet, comme :

- Classe Algorithme
- Classe Monte Carlo

Afin de mieux visualiser les relations entre les différentes classes présentées ci-dessus, le diagramme UML suivant synthétise leur organisation et leurs interactions. Ce schéma met en évidence les liens d’association entre les entités clés du projet ainsi que leur rôle dans la structure globale de la librairie.

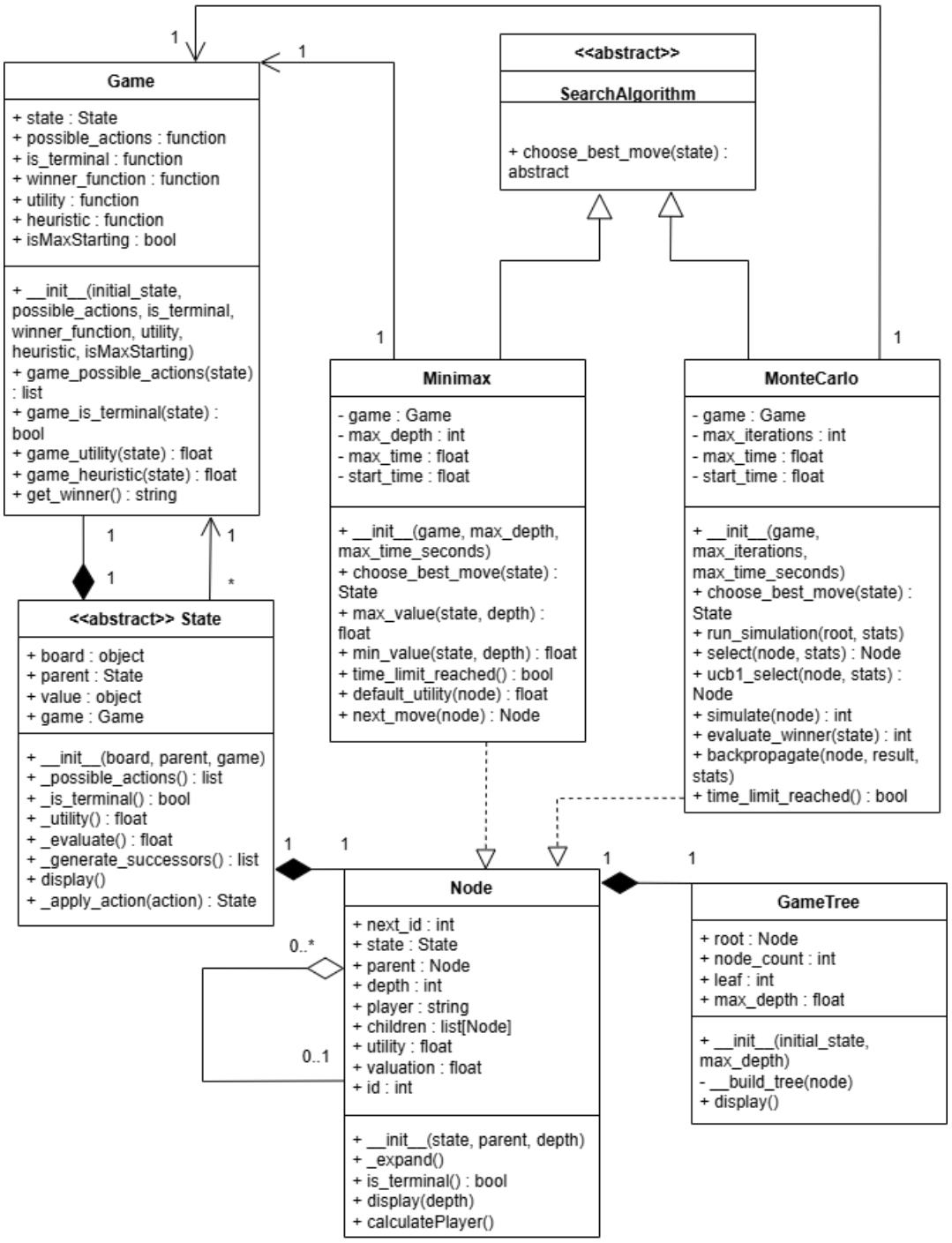


Figure 7. Diagram UML

4. Réalisation

Maintenant que la structure générale du projet est définie, il est temps d'entrer dans les détails de chaque composant. Nous commencerons par la classe Game, cœur de l'interface entre l'utilisateur et les algorithmes.

4.1 - Game

La classe Game joue un rôle fondamental dans notre architecture : elle représente l'ensemble de la logique du jeu, plus précisément les règles et les conditions de victoire propres à chaque jeu implémenté (Tic-Tac-Toe, Puissance 4, etc.).

Pensée comme un point d'entrée unique pour l'utilisateur, cette classe est conçue pour être la seule avec laquelle une personne souhaitant utiliser notre bibliothèque devra interagir directement. L'objectif est de séparer clairement la logique métier (les règles du jeu) du moteur de recherche (Minimax, Alpha-Beta, etc.), pour faciliter l'intégration de nouveaux jeux sans modifier l'algorithme.

Lors de l'instanciation, l'utilisateur doit fournir plusieurs fonctions spécifiques à son jeu :

- **initial_state** : l'état de départ du plateau ou du jeu.
- **possible_actions** : une fonction qui retourne toutes les actions légales à partir d'un état donné.
- **is_terminal** : une fonction qui vérifie si l'état actuel est un état final (fin de partie).
- **winner_function** : permet d'identifier le gagnant (ou une égalité).
- **utility** : évalue un état terminal pour déterminer l'issue de la partie.
- **heuristic** : évalue les états non terminaux pour guider la recherche (utilisé pour les arbres partiels).

Cette encapsulation permet à notre moteur d'interagir de façon générique avec n'importe quel jeu défini par l'utilisateur, tant que ces fonctions sont fournies. Autrement dit, la classe Game fait le lien entre la logique propre au jeu et les algorithmes de recherche.

Elle agit également comme un fournisseur de services pour les autres classes du projet, en exposant des méthodes simples telles que :

- game_possible_actions(state)
- game_is_terminal(state)
- game_utility(state)
- game_heuristic(state)
- get_winner()

Enfin, un paramètre isMaxStarting permet de définir si le joueur MAX commence la partie, ce qui peut avoir un impact sur le comportement initial de l'algorithme.

Pour rendre le moteur générique tout en évitant les appels redondants ou dispersés dans le code, nous avons choisi de centraliser les fonctions propres au jeu dans une instance unique de la classe Game.

```
    def __init__(self, initial_state=None, possible_actions=None, is_terminal=None,
                 winner_function=None, utility=None, heuristic=None, isMaxStarting=True):

        self.state = initial_state
        self.possible_actions = possible_actions
        self.is_terminal = is_terminal
        self.winner_function = winner_function
        self.utility = utility
        self.heuristic = heuristic
        self.isMaxStarting = isMaxStarting
        if self.state is not None:
            self.state.game = self
```

Les règles sont passées une seule fois au moment de l'instanciation (selon le principe d'injection fonctionnelle), puis partagées entre tous les composants via cette instance, évitant ainsi leur répétition dans chaque nœud ou état de l'arbre.

```
    def _is_terminal(self):
        return self.game.game_is_terminal(self)
```

Comme chaque Node contient un State, on accède indirectement à game :

```
    if self.is_terminal():
            self.utility = self.state._utility()
```

4.2 - Algorithme

Afin de pouvoir respecter la logique d'implémenter plusieurs algorithmes, nous avons défini une structure commune pour implémenter plusieurs algorithmes de recherche adverse (comme Minimax ou Monte Carlo). Elle permet de sélectionner dynamiquement l'algorithme à utiliser pour choisir le meilleur coup à partir d'un état donné, tout en évitant les problèmes d'import circulaire.

Pour que les algorithmes puissent manipuler les situations de jeu, il est essentiel de définir précisément ce qu'est un état. Voyons comment la classe State permet d'encapsuler ces informations.

4.3 - State

La classe State représente une position spécifique dans un jeu, indépendamment du contexte algorithmique dans lequel elle est utilisée. Un état encapsule l'ensemble des informations nécessaires pour décrire complètement la situation du jeu à un moment donné.

Les caractéristiques fondamentales :

Un État est défini par sa position de jeu, la configuration actuelle du plateau, des pièces, ou de tout autre élément constitutif du jeu. Cette position est autonome et n'est pas liée à la profondeur d'exploration dans l'arbre de recherche.

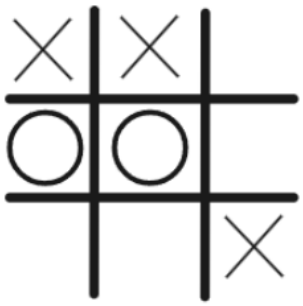
Chaque état possède la capacité de générer ses enfants en appliquant tous les coups possibles depuis sa position. Cette génération crée de nouveaux états correspondant à chaque action légale disponible. Les actions légales sont définies par l'utilisateur, puis passer à la classe game, c'est avec celle-ci que la classe etat vas interagir pour voir les action possible.

Un état peut être terminal, c'est-à-dire représenter une fin de partie.

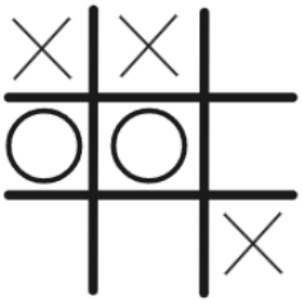
Distinction État vs Nœud :

Il est important de distinguer un état d'un nœud. Alors qu'un état représente une position de jeu pure, un nœud correspond à cette même position placée dans le contexte d'un arbre de recherche. Le nœud hérite des propriétés de l'état mais y ajoute des informations contextuelles : le joueur actuel, la profondeur dans l'arbre, et sa relation avec les nœuds parents et enfants dans la structure d'exploration. Il est simplement contextualiser dans l'arbre de jeux.

Voici un état



Voici un noeuds



```
utility : None
Heuristic : 0.2
Parent : 13252
Enfants : 13254
13255, 13256
depth : 7
Player : MIN
id : 13253
```

Figure 8. Noeuds vs Etat

Si un état décrit une configuration de jeu, un nœud (ou Node) l'inscrit dans un contexte algorithmique. Il permet notamment de structurer les décisions à travers un arbre de recherche. Voici comment cette distinction s'implémente dans notre projet.

4.4 - Nœud

La classe Node représente un élément de l'arbre de recherche dans le contexte des algorithmes de jeu. Contrairement à un état qui décrit une position pure, un nœud contextualise cette position au sein d'une structure d'exploration algorithmique.

Caractéristiques fondamentales d'un nœud :
Un nœud encapsule un état tout en y ajoutant des informations contextuelles essentielles pour la navigation dans l'arbre de recherche. Il maintient une référence vers l'état qu'il représente, mais enrichit cette information avec des métadonnées algorithmiques.

La hiérarchie est au cœur de la définition d'un nœud. Chaque nœud connaît son parent dans l'arbre (sauf la racine) et peut générer ses enfants via la méthode _expand(). Cette structure permet de maintenir le chemin complet depuis la racine jusqu'au nœud courant. La profondeur (depth) indique la distance du nœud par rapport à la racine de l'arbre. Cette information est cruciale pour les algorithmes avec limitation de profondeur et influence directement le calcul du joueur actuel.

Le joueur courant (player) est déterminé automatiquement en fonction de la profondeur et du joueur initial. Cette alternance systématique reflète la nature tour par tour des jeux adversaires.

Propriétés algorithmiques :
Chaque nœud possède un identifiant unique qui facilite le débogage et le suivi lors de l'exploration. Les valeurs d'utilité et d'évaluation heuristique sont calculées et stockées pour optimiser les performances des algorithmes.

La distinction fondamentale réside dans le fait qu'un nœud existe uniquement dans le contexte d'une recherche algorithmique, tandis qu'un état peut exister indépendamment de toute exploration. Le nœud apporte la dimension temporelle et relationnelle nécessaire aux algorithmes comme Minimax pour naviguer efficacement dans l'espace des possibilités.

Une fois les états et les nœuds bien définis, il devient possible de construire une structure globale qui relie tous les coups possibles : l'arbre de jeu. Celui-ci est central pour les algorithmes d'exploration comme Minimax.

4.5 - Arbre de Jeu
Un arbre de jeu est une structure de données hiérarchique qui représente l'ensemble des parties possibles d'un jeu à partir d'un état initial donné. Cette structure arborescente organise tous les scénarios de jeu envisageables, où chaque branche correspond à une séquence de coups alternés entre les joueurs.

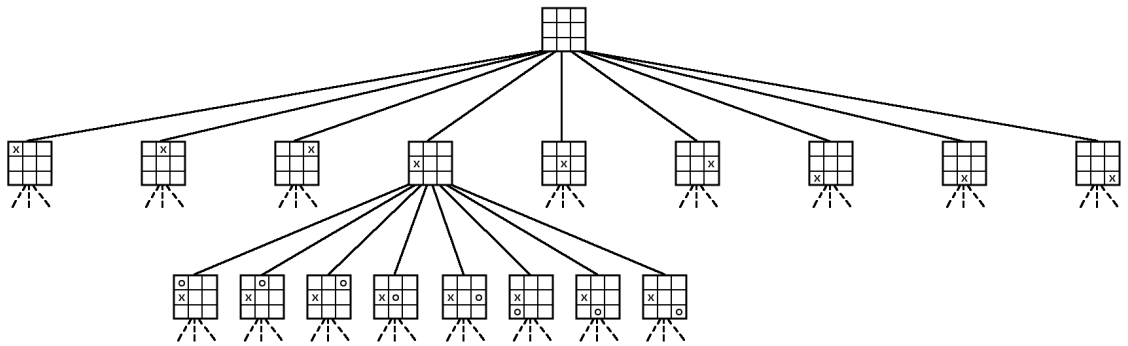


Figure 9. Arbre de jeux

Dans la théorie des jeux, un arbre capture la nature séquentielle et adversariale des décisions : chaque niveau représente un tour de jeu, et chaque nœud une position spécifique où un joueur doit prendre une décision. Les feuilles de l'arbre correspondent soit aux états terminaux naturels du jeu, soit aux positions où l'exploration s'arrête artificiellement selon des critères prédéfinis.

Utilisation dans le Contexte Algorithmique
La classe GameTree matérialise cette structure théorique pour les besoins des algorithmes de recherche adversariale. Elle sert de fondation aux algorithmes comme Minimax, qui nécessitent une exploration systématique de l'espace des possibilités pour déterminer la stratégie optimale.

Construction et paramétrage :
L'arbre est construit à partir d'un état initial qui devient la racine de la structure. Un paramètre de profondeur maximale (max_depth) permet de contrôler l'étendue de l'exploration, résolvant ainsi le problème d'explosion combinatoire inhérent aux jeux complexes.

Processus de génération :
La construction s'effectue de manière récursive via la méthode privée __build_tree(). À partir de la racine, l'algorithme explore en profondeur chaque branche possible, créant les nœuds enfants jusqu'à atteindre soit un état terminal (identifié par une valeur d'utilité définie), soit la limite de profondeur imposée.

Métriques et suivi :
L'arbre maintient des statistiques sur sa propre structure : le nombre total de nœuds explorés (node_count) et le nombre de feuilles générées (leaf). Ces métriques sont essentielles pour évaluer la complexité de l'exploration et optimiser les performances des algorithmes qui opèrent sur cette structure. Cela sera surtout utilisé lors des tests et debugs.

L'arbre de jeu constitue ainsi le socle sur lequel reposent les algorithmes de décision, transformant l'abstraction théorique du jeu en une structure de données concrète et explorable algorithmiquement.

Sur cette base structurelle, nous pouvons maintenant implémenter notre premier algorithme d'exploration adverse : Minimax. Simple, mais puissant, il constitue une étape idéale pour commencer notre expérimentation algorithmique.

4.6 - Minimax
L'algorithme Minimax suit une méthode systématique que le document décompose clairement. Il commence par générer l'arbre complet de tous les coups possibles jusqu'aux états terminaux. Une fois cet arbre construit, il applique la fonction de récompense à toutes les feuilles pour les étiqueter avec leur valeur respective.

Cette évaluation des feuilles est représentée par trois valeurs possibles, que l'on appelle la valeur d'utilité.

- Si victoire la valeur est un 1
- Si défaite la valeur est un -1
- Si nulle la valeur est un 0

Vient ensuite la phase de remontée, où les valeurs propagent depuis les feuilles vers la racine. À chaque niveau, l'algorithme étiquette chaque nœud avec le meilleur résultat possible pour le joueur concerné. Enfin, cette propagation permet de construire la stratégie en identifiant les meilleures actions pour le joueur MAX à chaque étape.

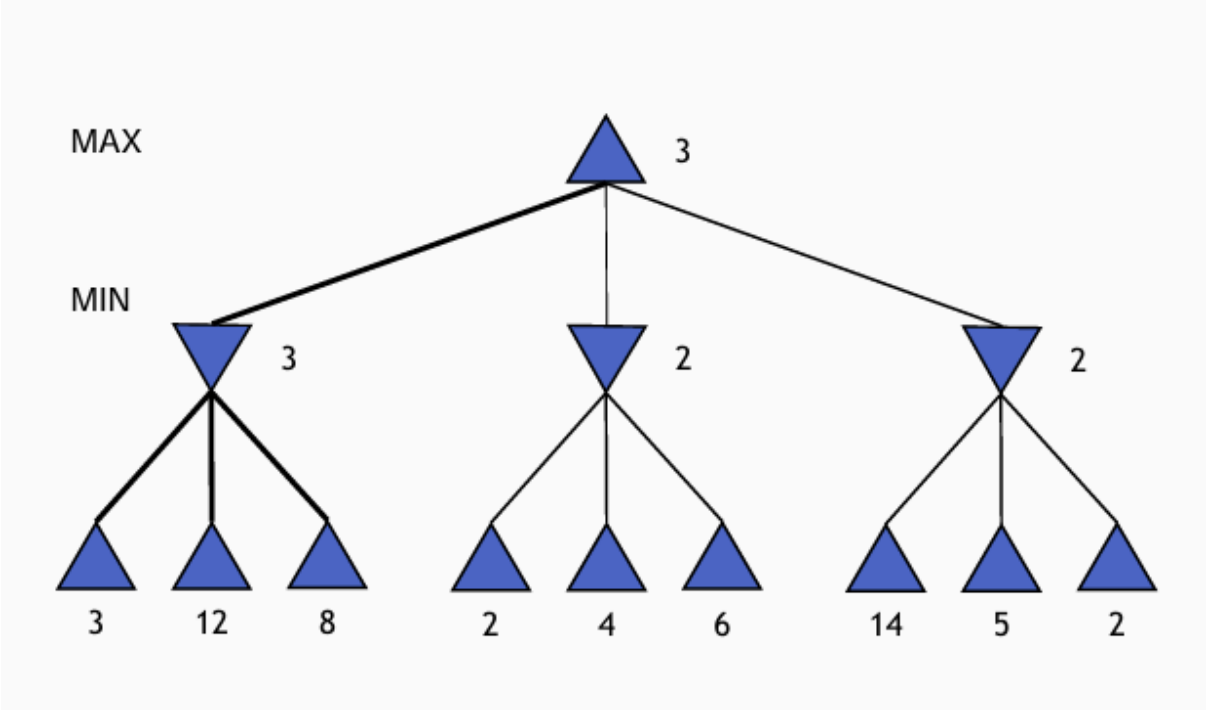


Figure 10. Propagation des valeurs

Pour que l'algorithme Minimax fonctionne correctement, le document explique qu'il faut partir de deux hypothèses importantes concernant l'adversaire. Premièrement, on considère que l'opposant est rationnel, c'est-à-dire qu'il prend ses décisions de manière logique et cohérente. Deuxièmement, on suppose qu'il joue de façon optimale, en choisissant toujours les meilleurs coups possibles pour maximiser ses chances de victoire. Ces deux hypothèses sont essentielles car elles permettent à Minimax de prédire le comportement de l'adversaire et donc de calculer la meilleure stratégie à adopter.

Afin de respecter ces hypothèses, nous allons utiliser une évaluation pour chaque nœud. Cette évaluation est représentée par des chiffres négatifs ou positifs. Ainsi, pour que chaque joueur soit pertinent, MAX valorisera la plus haute valeur et MIN la plus basse valeur.

Le Problème de la Profondeur
La première limitation majeure identifiée dans le document concerne l'impossibilité d'explorer jusqu'aux feuilles terminales dans la plupart des jeux réels. Pour résoudre ce problème, on utilise des tests de coupure qui définissent une limite de profondeur ou d'autres critères d'arrêt. Cela oblige à générer un arbre partiel où les feuilles ne sont pas des états terminaux naturels.

Pour illustrer ce concept, le document prend l'exemple concret des échecs. Avec environ 35 coups possibles à chaque tour et des parties qui peuvent atteindre 100 coups de profondeur, l'exploration complète devient rapidement impossible. Ces chiffres montrent pourquoi Minimax pur ne peut pas être utilisé directement sur des jeux complexes.

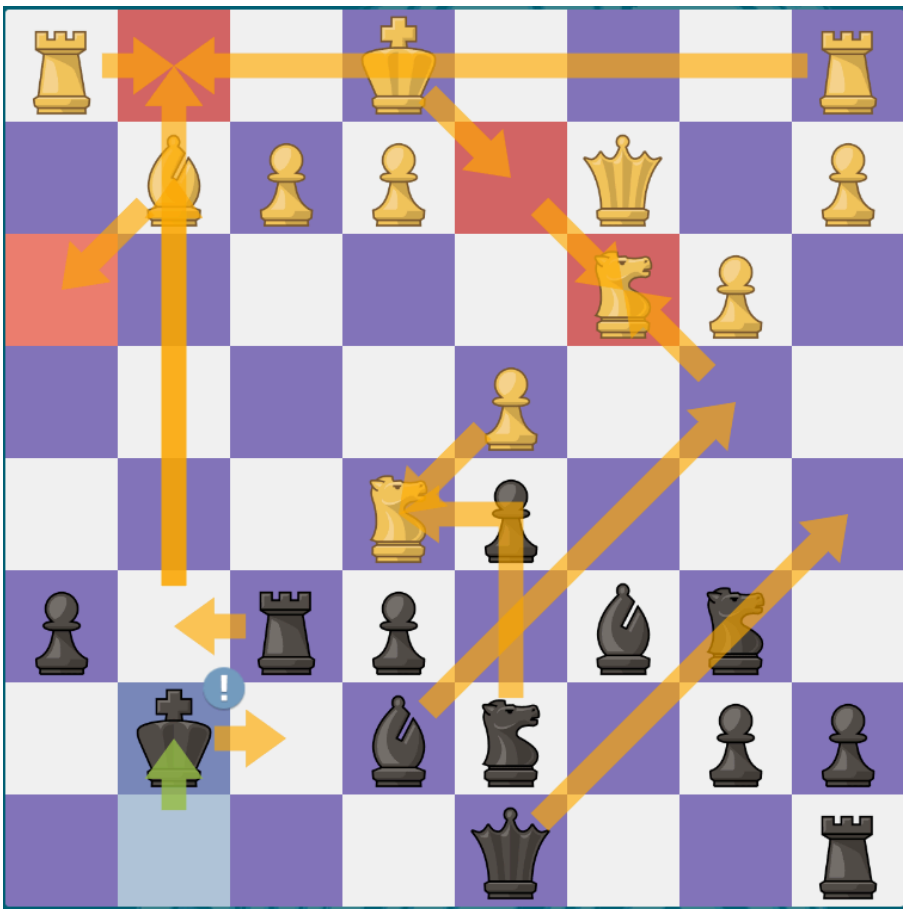


Figure 11. Démonstration de la profondeur des Échec

Dans ce cas, on applique une fonction heuristique à chaque feuille artificielle en supposant qu'elle représente fidèlement la vraie fonction de récompense. L'algorithme Minimax fonctionne ensuite normalement sur cet arbre tronqué, mais sa qualité dépend entièrement de la pertinence de l'heuristique utilisée.

L'heuristique utilisée dans notre projet est conçue pour être améliorée progressivement au fil du développement. Elle dépend fortement du jeu spécifique sur lequel l'algorithme est appliqué. C'est pourquoi nous avons choisi de laisser à l'utilisateur la responsabilité de fournir sa propre fonction heuristique, afin de garantir que notre librairie reste générique et adaptable à tout jeu à somme nulle.

Mais dans une optique de test, nous avons également ajouté une limite de profondeur. La profondeur représente le nombre de coups joués. Plus nous augmentons cette variable, plus nous devons descendre dans l'arbre de décisions. Ainsi, lors de certains tests, une profondeur suffisante pour atteindre les premières feuilles permet un test plus rapide.

L'algorithme Minimax s'exécute de la manière suivante, à partir d'un état initial, passé en paramètre, nous utilisons la classe State pour générer tous les états enfants possibles, correspondant aux coups jouables.

Selon le joueur courant, deux fonctions distinctes sont appelées :

- max_value, qui cherche à maximiser la valeur (pour le joueur MAX),
- min_value, qui cherche à minimiser la valeur (pour le joueur MIN).

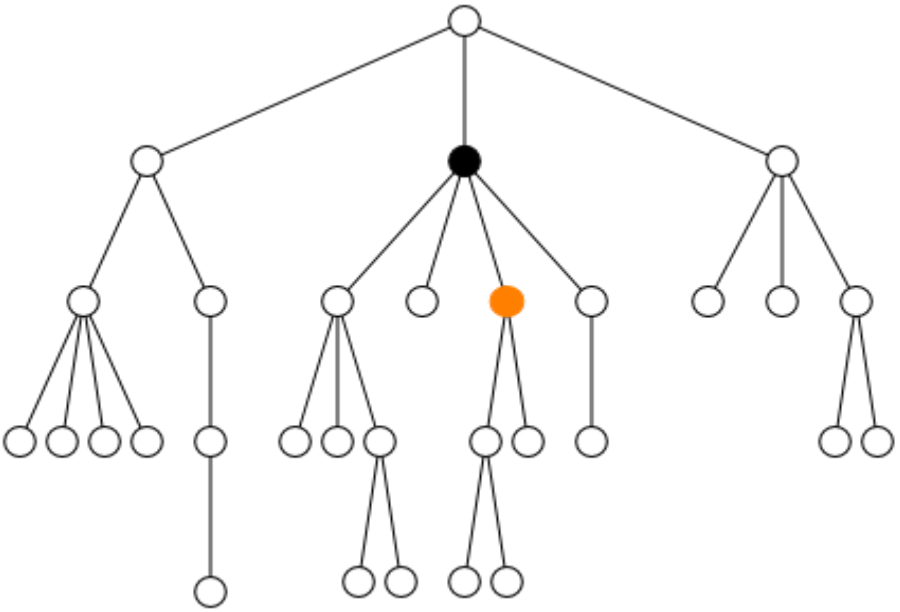


Figure 12. Enfant et parent dans un arbre

Chaque enfant est ensuite évalué. Nous commençons par vérifier s'il s'agit d'un état terminal (c'est-à-dire une feuille de l'arbre). Cette vérification repose sur la valeur d'utilité (utility) retournée :

- Si aucune valeur n'est définie, l'état n'est pas terminal.
- Si une valeur d'utilité est présente (1, -1 ou 0), alors l'état est une feuille.

Selon cette valeur et le joueur actuel, plusieurs cas se présentent :

- Utility = 1 :
  - Si le joueur est MAX (dans max_value), cet enfant sera favorisé.
  - Si le joueur est MIN (dans min_value), il cherchera à éviter ce coup.
- Utility = -1 :
  - Si le joueur est MIN, il privilégiera cet enfant.
  - Si le joueur est MAX, il l'évitera.

- Utility = 0 (égalité) :
    - L'algorithme continue d'évaluer les autres enfants.
    - Si aucun coup n'est plus avantageux, alors l'égalité sera acceptée comme meilleur choix.

A cette vérification, si aucun enfant a été choisi via sa valeur utility, alors nous évaluons la valeurs heuristique de tous les enfants qui de sont pas des feuilles. afin de savoir quels enfants est le plus optimal, selon le joueur.
Cette enfant est donc renvoyer comme le meilleur coup dans la position (l'état) donné en paramètre.

Exemple avec le tic tac toe :

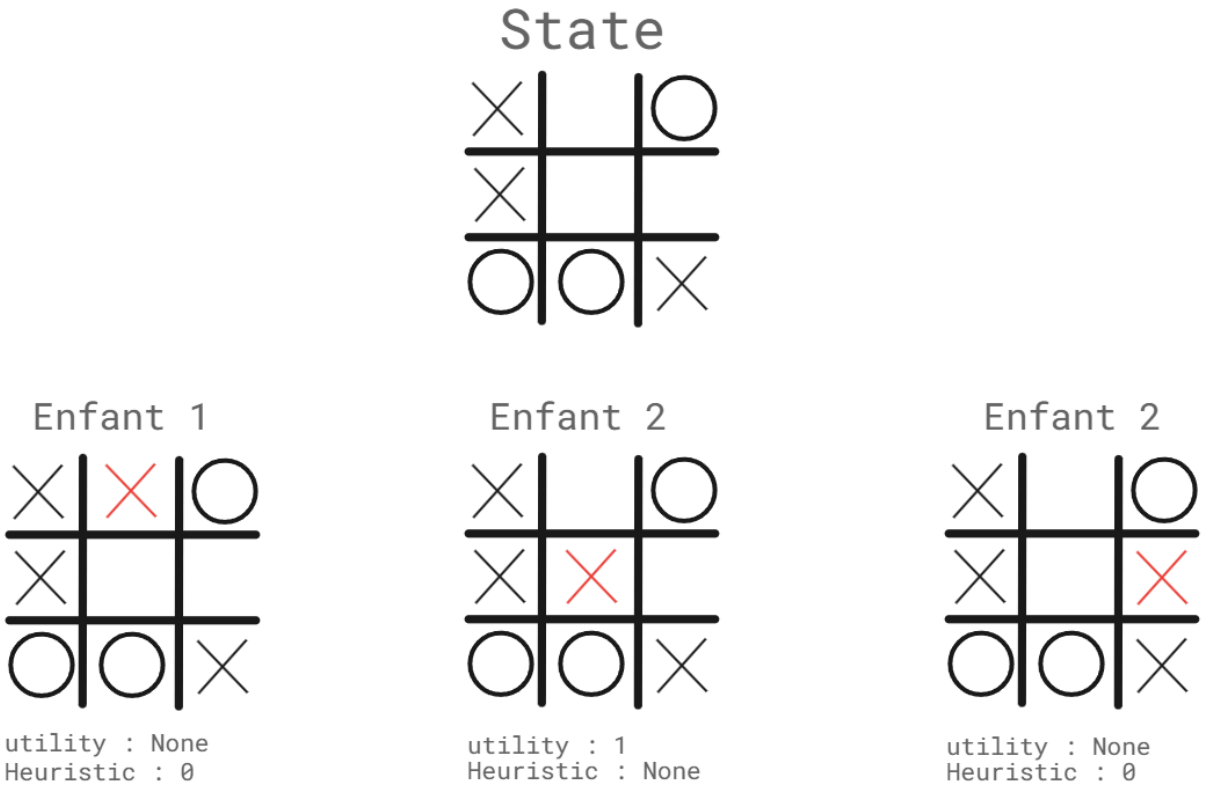


Figure 13. Exemple fonctionnement Minimax

Ici, nous avons un état de départ qui possède trois enfants. Deux enfants ont pour valeur heuristique 0. Nous avons considéré que l'utilisateur obtient ce résultat du fait que toutes les feuilles précédant ces enfants sont des matchs nuls. Ainsi, nous avons un troisième enfant avec une valeur d'utilité de 1, ce qui correspond à une victoire pour le joueur MAX, actuellement en train de jouer. C'est donc cet enfant qui va ressortir comme le meilleur coup à jouer depuis l'état donné.

4.7 - Monte carlo

Suite au succès de Minimax, notre groupe a pris la décision d'avancer sur un autre algorithme. Je me suis alors porté volontaire pour effectuer les premières recherches et le développement de l'algorithme Monte Carlo.

L’algorithme Monte Carlo repose sur une approche probabiliste qui se distingue fortement de la méthode systématique utilisée par Minimax. Plutôt que d’explorer tout l’arbre de manière exhaustive, Monte Carlo s'appuie sur un grand nombre de simulations aléatoires pour estimer la qualité des coups possibles à partir d’un état donné.

Pour chaque coup possible depuis l’état courant, l’algorithme simule un grand nombre de parties complètes en jouant au hasard jusqu’à la fin du jeu. En observant les résultats de ces simulations (victoires, défaites, matchs nuls), il estime la qualité de chaque coup par fréquence de victoire. Le meilleur coup est celui qui, en moyenne, mène le plus souvent à la victoire.

Dans Monte Carlo, chaque simulation est une partie jouée jusqu’à un état terminal, où les actions sont choisies de façon aléatoire. À la fin de chaque simulation, on obtient la valeur utility (les mêmes que pour minimax):

- 1 si le joueur MAX gagne
- -1 si c’est le joueur MIN
- 0 si la partie se termine par une égalité

Ces résultats sont ensuite utilisés pour mettre à jour les statistiques associées aux coups testés. Plus un coup a mené à des victoires lors des simulations, plus sa valeur estimée augmente.

Monte Carlo suit un cycle en quatre étapes, répété des milliers de fois :

Sélection
À partir de l’état de départ, l’algorithme choisit un nœud de l’arbre partiellement construit. Dans la version simple, ce choix peut être aléatoire ou reposer sur une stratégie plus intelligente. ici nous allons utiliser l'algorithme UCB1, méthode utilisée pour choisir entre plusieurs options (par exemple, plusieurs coups dans un jeu), en cherchant à trouver un bon équilibre entre :

- Exploiter ce qu’on connaît déjà (choisir les coups qui ont bien marché jusqu’à présent),
- Explorer ce qu’on connaît peu (tester des coups moins joués, qui pourraient être meilleurs).

Plutôt que de toujours prendre le meilleur coup actuel, UCB1 donne aussi une chance aux coups moins explorés. Il attribue un score à chaque coup qui combine sa performance passée et le nombre de fois qu’il a été testé.

Expansion
Si le nœud sélectionné n’est pas terminal, l’algorithme génère un ou plusieurs enfants, correspondant à des coups possibles. Un de ces enfants est choisi pour la simulation.

Simulation (Playout)
À partir de cet enfant, une partie est simulée en jouant au hasard jusqu’à un état terminal.

Rétropropagation (Backpropagation)
Le résultat de la simulation est remonté à travers les nœuds visités pour mettre à jour les statistiques (nombre de simulations, nombre de victoires, etc.).

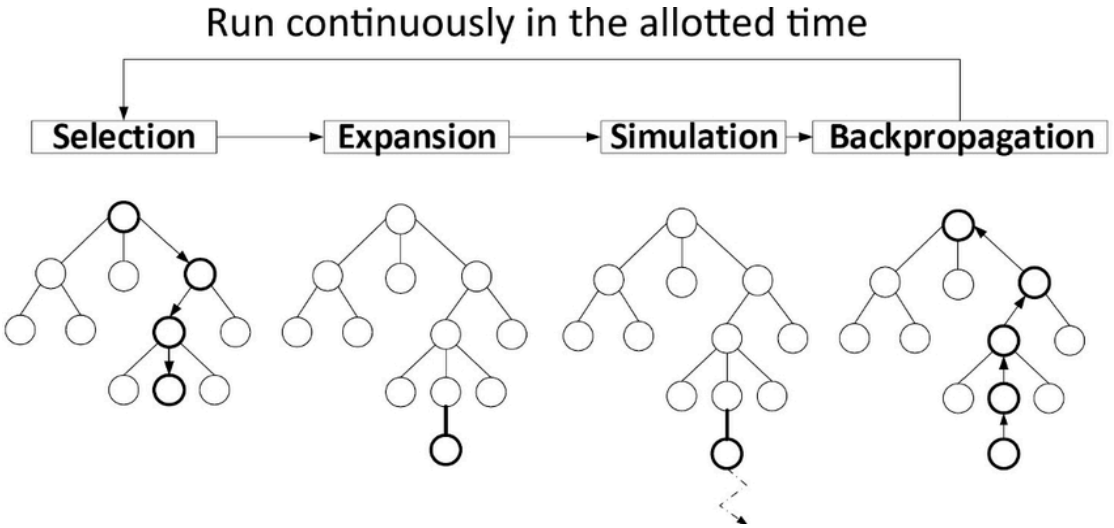


Figure 14. Les quatres étapes de Monte Carlo

Voici comment cela se représente dans mon code.

```
def run_simulation(self, root, stats):
    node = self.select(root, stats)                # Étape 1 : Sélection
  #-------------------------------------------------
    if not self.game.game_is_terminal(node.state):
        node._expand()                             # Étape 2 : Expansion
        if node.children:
            node = random.choice(node.children)
  #-------------------------------------------------
    result = self.simulate(node)                   # Étape 3 : Simulation (Playout)
  #-------------------------------------------------
    self.backpropagate(node, result, stats)        # Étape 4 : Rétropropagation
```

Sélection :

```
def select(self, node, stats):
    while node.children:

        node = self.ucb1_select(node, stats)
    return node

def ucb1_select(self, node, stats):
    total_visits = sum(stats.get(child.id, (0, 1))[1]
                       for child in node.children)
    log_total = math.log(total_visits + 1)
    best_score = -float('inf')
    best_child = None

    for child in node.children:
        wins, visits = stats.get(child.id, (0, 1))
        win_rate = wins / visits
        ucb1 = win_rate + math.sqrt(2 * log_total
                                    / visits)
        if ucb1 > best_score:
            best_score = ucb1
            best_child = child

    return best_child
```

La méthode select() effectue cette phase de sélection en descendant dans l'arbre jusqu'à un nœud feuille. Et c'est dans ucb1_select() que s'applique l'algorithme UCB1 pour choisir les enfants les plus prometteurs, en équilibrant exploration et exploitation.

Expansion :

```
if not self.game.game_is_terminal(node.state):
    node._expand()                          # Étape 2 : Expansion
    if node.children:
        node = random.choice(node.children)
```

Après la sélection, si le nœud n'est pas terminal, tu appelles _expand() pour générer les enfants. Puis tu choisis aléatoirement l'un d'entre eux pour simuler la suite.

Simulation :

```
def simulate(self, node):
    state = node.state
    while not self.game.game_is_terminal(state):
        actions = state._possible_actions()
        if not actions:
            break
        action = random.choice(actions)
        state = state._apply_action(action)
    return self.evaluate_winner(state)
```

Ici, la simulation joue au hasard jusqu'à un état final, puis évalue le gagnant.

Rétropropagation :

```
def backpropagate(self, node, result, stats):
    while node is not None:
        if node.id not in stats:
            stats[node.id] = (0, 0)
        wins, visits = stats[node.id]
        stats[node.id] = (wins + result, visits + 1)
        node = node.parent
```

Chaque nœud du chemin vers la racine voit ses statistiques mises à jour : nombre de visites, nombre de victoires.

Contrairement à Minimax, Monte Carlo ne suppose pas que l'adversaire joue de manière optimale. Il suppose simplement que le comportement global du jeu peut être approximé par des essais aléatoires massifs. C'est ce qui fait sa force dans les environnements complexes, où Minimax devient rapidement inapplicable à cause de la profondeur.

De plus, Monte Carlo ne nécessite pas de fonction d'évaluation heuristique si les simulations vont jusqu'au bout. Cela le rend plus générique, car il n'a pas besoin d'être spécifiquement adapté à chaque jeu, à condition qu'il soit possible de simuler des parties rapidement.

4.8 - Les Test

Les objectif de test, valider le bon fonctionnement des algorithmes d'exploration adverse, en particulier Minimax dans un premier temps. Observer et analyser leur comportement sur un jeu concret, ici le Tic-Tac-Toe, dans différentes conditions de jeu

Deux fonctions de test principales sont proposées dans le fichier test_minimax.py :

- test_minimax_via_algorithm_selector(): une simulation automatique entre deux IA (MAX vs MIN).
- test_minimax_via_algorithm_selector_human_player(): permet à un joueur humain d'affronter l'IA utilisant Minimax.

Voici un exemple de partie jouée contre l'algorithme Minimax. Je joue la croix, Minimax joue le cercle.

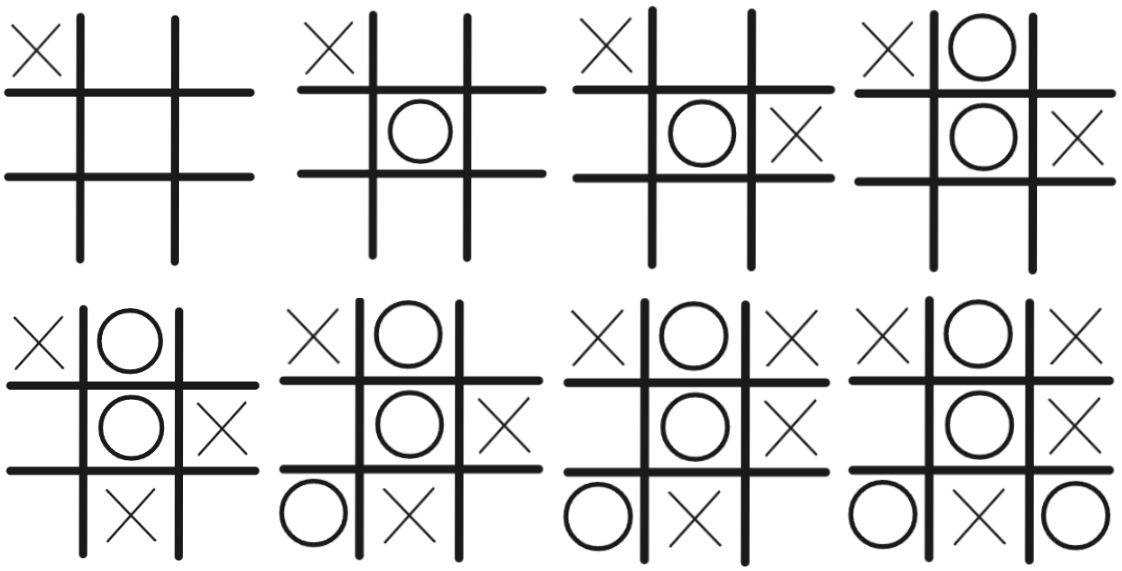


Figure 15. Partie de test Minimax

Et c'est évidemment un match nul.

Chaque test utilise une version personnalisée du jeu Tic-Tac-Toe, fournie par la fonction generate_tictactoe_game().

Cette fonction provient d'un fichier servant de définition pour toutes les fonctions qui doivent être passées dans la classe Game, notamment la fonction heuristique. Nous avons vu plus tôt à quoi elle sert, nous allons maintenant voir comment elle fonctionne.

Heuristique
Nous avons utiliser une formule,
h(s) = 0.5 × (# de 2-en-ligne pour MAX)
- 0.5 × (# de 2-en-ligne pour MIN)
+ 0.2 × (bonus si X est au centre)
- 0.2 × (malus si O est au centre)
+ 0.1 × (nombre de cases vides restantes)

2-en-ligne, représente la menace ou l'opportunité de gagner bientôt, cela veux dire que deux signes d'un même joueur sont alignés.
L'algorithme compte combien de lignes, colonnes ou diagonales contiennent exactement :

- 2 symboles 'X' et 1 case vide ➜ potentielle victoire proche de MAX.
- 2 symboles 'O' et 1 case vide ➜ menace de MIN qu'il faut bloquer.

Ceci également pour MIN.

```
two_max = sum(1 for line in lines if line.count('X') == 2 and line.count(' ') == 1)
two_min = sum(1 for line in lines if line.count('O') == 2 and line.count(' ') == 1)
```

Le centre (case b[1][1]) est stratégiquement la plus forte au début de la partie. Le joueur qui occupe le centre peut participer à plus de combinaisons gagnantes. C'est pourquoi l'heuristique ajoute ou retire des points selon que le centre est occupé par X ou O :

```
center_val = 1 if b[1][1] == 'X' else -1 if b[1][1] == 'O' else 0
```

et pour finir, plus il reste de coups à jouer, plus l'espace stratégique est ouvert.

```
diff_mobility = sum(1 for i in range(3) for j in range(3) if b[i][j] == ' ')
```

Cela favorise des états non bloqués, où MAX garde un bon contrôle du plateau. Chaque case vide ajoute +0.1.

Exemple de notre heuristique :

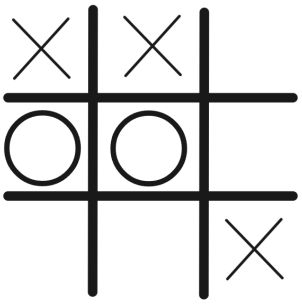


Figure 16. État d'un tic tac toe

Analyse :

- X a 1 ligne avec 2 symboles et 1 vide ➜ +0.5
- O a 1 ligne avec 2 symboles et 1 vide ➜ -0.5
- Centre (b[1][1]) = O ➜ -0.2
- Cases vides : 4 ➜ +0.4

Alors nous avons ce calcule finial

- 0.5 - 0.5 - 0.2 + 0.4 = 0.2

Monte Carlo

Voici une partie jouée avec la première version de l'algorithme Monte Carlo implémentée dans notre projet.

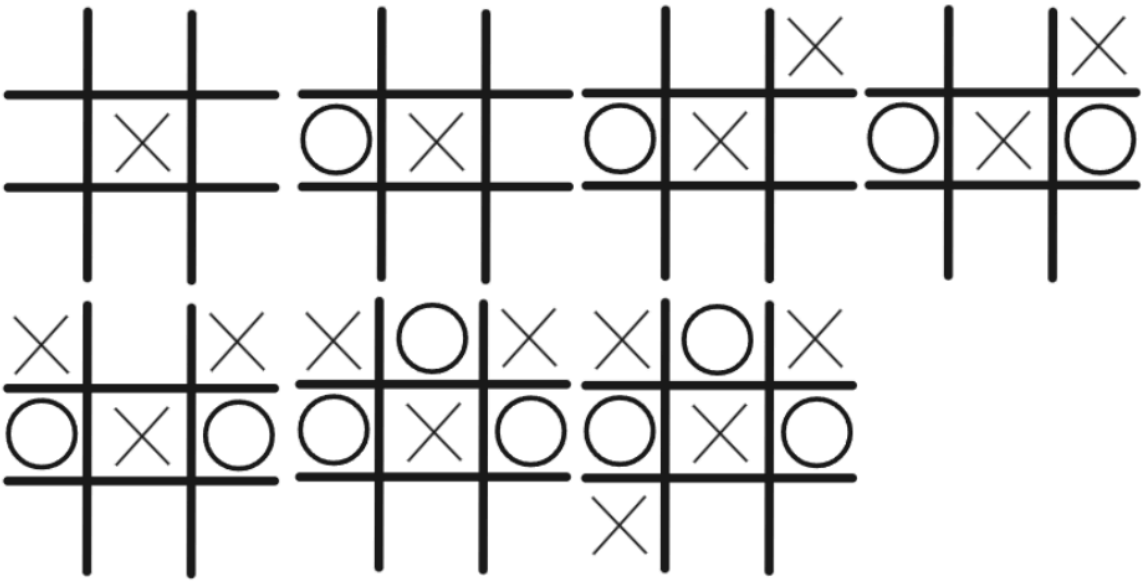


Figure 17. Partie Monte Carlo

Comme nous pouvons le constater, cette première version de l'algorithme Monte Carlo présente une faiblesse importante : il manque un coup qui lui aurait permis de gagner immédiatement. Ce comportement s'explique par le fonctionnement même de l'algorithme. Comme évoqué précédemment, Monte Carlo évalue les coups possibles en simulant un grand nombre de parties aléatoires à partir de chacun d'eux, puis sélectionne celui qui obtient statistiquement le plus de victoires à long terme.

Dans ce cas précis, au quatrième coup, au lieu de choisir une action qui aurait immédiatement conduit à une victoire, l'algorithme a préféré un autre coup qui menait à plus de feuilles "gagnantes" au total dans l'arbre de simulation, sans tenir compte du fait que l'autre coup offrait une victoire immédiate.

Ce test met également en lumière une autre limite : l'absence de stratégie défensive. En effet, l'algorithme ne prend pas en compte les menaces immédiates de l'adversaire, car il se base uniquement sur la fréquence de ses propres victoires. Il peut ainsi ignorer des coups critiques nécessaires pour bloquer l'adversaire.

Une piste d'amélioration pourrait consister à introduire un mécanisme de reconnaissance des états terminaux immédiats avant le lancement des simulations. Par exemple, on pourrait prioriser les coups menant directement à une victoire ou bloquant une défaite imminente, avant de recourir aux simulations probabilistes. Cela permettrait de combiner la rigueur tactique de Minimax avec la flexibilité de Monte Carlo.

4.9 - Outils de visualisation

Afin de mieux comprendre et illustrer le fonctionnement des algorithmes adverses comme Minimax, un outil de visualisation graphique a été conçu. Celui-ci permet de naviguer pas à pas dans l'arbre d'exploration, à partir d'un état initial, et de visualiser dynamiquement les nœuds visités et les connexions créées.

Cette visualisation apporte une compréhension intuitive du parcours de l'arbre, particulièrement utile lors de la mise au point des fonctions d'expansion, de sélection ou d'évaluation.

Structure générale de l'outil

L'outil est constitué de deux classes principales :

- Visualization : gère l'interface graphique et la logique de navigation.
- Node : représente un nœud de l'arbre, incluant son état, ses enfants, et sa profondeur.

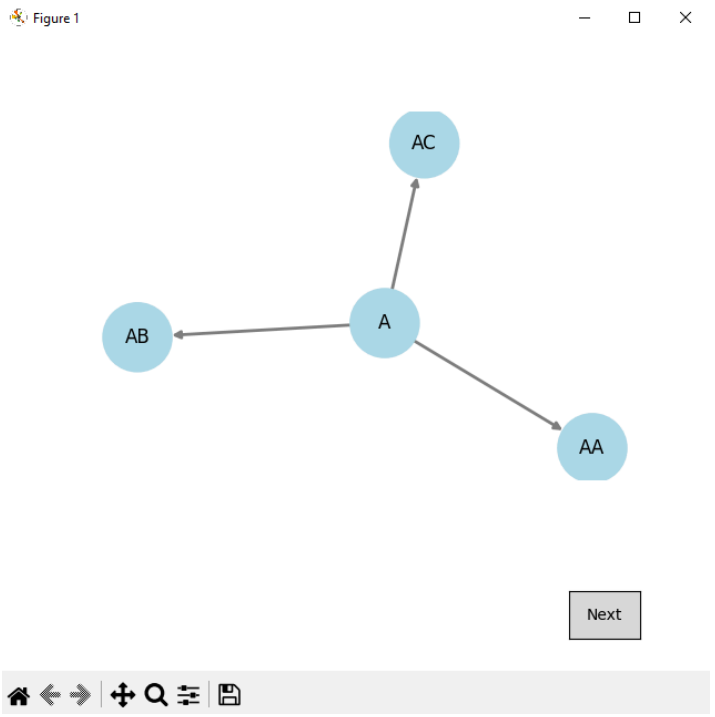


Figure 18. Outils de visualisation

Chaque nœud représente un état de jeu ou une configuration simulée, et les arêtes indiquent les transitions possibles (actions).

Initialisation d'un état simulé
L'exemple utilisé dans la visualisation repose sur une classe GameState simplifiée. Chaque état contient un nom et peut générer trois actions tant qu'une profondeur maximale n'est pas atteinte.

```
class GameState:
    def __init__(self, name, depth=3):
        self.name = name
        self.depth = depth
        if self.depth > 0:
            self.actions = [name + c for c in "ABC"]
        else:
            self.actions = []

    def possible_actions(self):
        return self.actions

    def apply_action(self, action):
        return GameState(action, self.depth - 1)

    def is_terminal(self):
        return len(self.actions) == 0

    def __str__(self):
        return self.name
```

Ce système permet de générer un arbre de manière déterministe et contrôlée, facilitant ainsi la visualisation.

Représentation d'un nœud
Chaque état est encapsulé dans un objet Node, qui contient un lien vers son parent, sa profondeur, et ses successeurs.

```
class Node:
    def __init__(self, state, parent=None, depth=0):
        self.state = state
        self.parent = parent
        self.depth = depth
        self.children = []

    def expand(self):
        if not self.children:
            for action in self.state.possible_actions():
                new_state = self.state.apply_action(action)
                self.children.append(Node(new_state, parent=self, depth=self.depth + 1))
```

Affichage et navigation
La classe Visualization utilise matplotlib et networkx pour dessiner dynamiquement l'arbre. À chaque clic sur un bouton "Next", un nouveau nœud est sélectionné et affiché. Comme montré dans la figure outils de visualisation.

```
def next_node(self, event):
    if not self.current_node.children:
        self.current_node.expand()
    if self.current_node.children:
        self.current_node = self.selection_method(self.current_node.children)
        self.history.append(self.current_node)
        self.visited_nodes.add(self.current_node)
        self.draw_graph()
```

5. Les difficultées dans l'adversité

En dehors des défis techniques, le projet a aussi révélé des difficultés organisationnelles liées à la diversité des profils dans l'équipe. Cette section en dresse un aperçu honnête.

Je vous ai déjà, bien plus haut, présenté les membres de mon équipe. Comme je l'ai précisé, certains ne sont pas en informatique et ne maîtrisent donc pas la programmation, même un peu.

Et c'est bel et bien sur ces documents, ces tâches, que nous apercevons ce problème, car l'attribution des tâches a été faite selon plusieurs critères, et l'un d'eux était bien entendu la capacité de développement. C'est pourquoi les documents tels que le ReadMe étaient attribués à ceux sans expérience en programmation.

La moitié du groupe a été attribuée à ces tâches, ce qui, sur le papier, ne semble pas une mauvaise chose, puisque plus de personne signifie moins de temps à consacrer à ces tâches, en plus de permettre une mise à jour régulière tout au long du projet.

Malheureusement, le travail fourni sur cette tâche a été inégal : trois personnes ont été attribuées au document, mais seulement deux ont travaillé dessus. Lors de sa relecture, le document a dû être réécrit à 80 % par un développeur de l'équipe.

6. Enseignement en mémoire

Nous avons pu en retirer beaucoup de positif. Le premier point, paradoxalement, concerne la collaboration avec les membres non-développeurs du groupe. Il était très compliqué de leur attribuer des tâches avec un minimum de complexité. Alors, quand leur détermination le permettait, des sessions de peer coding étaient mises en place afin d'expliquer, lors de la réalisation des tâches, le code généré par les programmeurs, pour avoir une meilleure vision du projet. Ainsi, malgré les difficultés, l'un des membres non-développeurs a pu suivre les sessions de peer coding et, à la fin de son stage, a acquis une très bonne compréhension du projet ainsi que quelques bases en programmation orientée objet.

Pour rester dans la collaboration, il a été très instructif de travailler avec les membres développeurs du projet. Simplement parce que nos niveaux étaient inégaux eux aussi, nous avons donc pu apprendre les uns des autres. Le Trello qui avait été mis en place a très rapidement été abandonné, car personne n'avait l'habitude de l'utiliser. Il a été remplacé par une simple page de rapport que je devais fournir au Boss chaque semaine.

Tâche de la semaine 19 février 2025

Tâche	Membre	Durée	Priorité
renommer la fonction « evaluate » en « utility » (feuille) et la récupérer dans le module game. Il doit également créer dans game une fonction is_terminal qui prend un état en paramètre et sera utilisée sous la forme « etat.leJeu.is_terminal(self) », où self représente l'état courant.	Ben	5h	Haute
modifier la variable « Node » en « node » afin d'assurer l'uniformité avec le reste du fichier dans l'algorithme minimax, et il doit également implémenter l'évaluation directement dans les noeuds.	Ewan	4h	Haute
chargé de tester si la référence d'objet est passée en Python ou si l'objet est dupliqué. Il doit par ailleurs ajouter dans tools les attributs pour un noeud, notamment la profondeur, l'ordre dans cette profondeur ainsi qu'une valuation (allant de 2 à 5).	Hugo	4h	Haute
rédiger une ébauche de README expliquant le but du projet et les principes des algorithmes, en se concentrant pour l'instant sur l'algorithme minimax. Ce document devra comporter une partie générale, une section dédiée aux tools et une section pour les classes.	Mathis, Axel et Anais	2h	Basse
intégrer une référence au module game dans l'algorithme minimax et renommer la variable « Node » en « node », afin qu'elle soit considérée comme une variable et non comme une classe. dans le fichier game	Martin et Anais	4h	Haute

Figure 19. Tâches de la semaine

De plus, le rôle du destructeur a été crucial : il nous a permis de remettre en question notre façon de faire, et de réparer, éclaircir et optimiser notre code. J'ai pu citer plus tôt l'exemple suivant : lors de la création de la classe Game, le destructeur s'est rendu compte que nous appelions les fonctions autant de fois que le nombre de nœuds (250 000 pour un Tic Tac Toe), ce qui pouvait causer une utilisation accrue des ressources et un temps plus long pour créer l'arbre. C'est pourquoi le choix de centraliser les fonctions propres au jeu dans une instance unique de la classe Game, ainsi que le principe d'injection fonctionnelle, nous est venu dans l'optique d'optimisation.

VI. Perspective

Malgré ces difficultés, le projet a ouvert de nombreuses pistes d'amélioration et de développement futur, tant du point de vue technique que collaboratif.

Ce stage est orienté vers la recherche, les possibilités et les perspectives d'avenir sont extrêmement nombreuses. Mais nous avons déjà entrevu certaines de ces possibilités.

Tout d'abord, nous devons terminer les différents algorithmes commencés. Monte Carlo est très loin d'avoir une précision acceptable. De plus, l'algorithme Alpha-Beta doit être mis en place. Nous pouvons également envisager qu'une plus large gamme d'algorithmes soit disponible.

Nous pourrions aussi ajouter une capacité de mémoire pour les algorithmes qui nécessitent un entraînement, comme Monte Carlo, afin que l'utilisateur puisse récupérer un algorithme entraîné sur le long terme.

Le point le plus évident est de créer une librairie Python. Maintenant que le code est produit et fonctionnel, il va de soi que la prochaine étape est sa publication, via GitHub actuellement, mais également, à terme, sous la forme d'une librairie Python.

La documentation doit être mise à jour continuellement. C'est donc également un point qui devra être pris en compte à l'avenir.

VII. Bilan

Ce stage m'a permis d'affiner mes connaissances dans un domaine comme l'algorithmique. J'ai pu mettre en parallèle ce travail avec un cours d'algorithmique effectué à la même période. Cela m'a donné une pratique que je ne retrouvais pas dans les cours plus théoriques. Cela m'a offert un autre point de vue sur l'utilisation des différents algorithmes, comme Monte Carlo, étudié en cours.

D'un point de vue collaboration, ce stage m'a beaucoup apporté, notamment la collaboration entre développeurs pour réaliser les tâches, mais aussi avec les non-développeurs afin de simplifier le propos et garder une vision claire du projet. Devoir expliquer à un novice nous fait souvent prendre du recul sur le projet.

D'un point de vue purement programmation, nous avons pu nous rendre compte de l'importance de l'optimisation dans la recherche. Dans un domaine où l'on conçoit des centaines de milliers d'opérations, le moindre problème d'optimisation peut nous coûter un temps énorme. Nous avons également dû penser à l'utilisateur, même si le contact avec celui-ci est moins important que dans une application classique, il était important de garder à l'esprit que nous développons dans un but bien précis, au service de l'utilisateur.

1. Conclusion

Ce stage est mon deuxième dans le domaine de la recherche. Il m'a permis d'observer les différences de méthodes de travail entre la France et le Canada. Cette richesse d'apprentissage m'offre aujourd'hui la capacité de m'adapter plus facilement aux environnements professionnels variés que je pourrai rencontrer dans ma carrière.

Travailler concrètement dans le domaine de la programmation algorithmique m'a également permis de me réconcilier avec le cours d'algorithmique, en donnant du sens pratique aux concepts appris au détriment de quelques nuits de sommeil.

Les relations humaines durant le projet ont été très agréables et fluides. Pour cela, je remercie encore et toujours mon maître de stage, Monsieur Kevin Bouchard.

VIII. Résumé en anglais

My name is Ewan Burasovitch, I'm 20 years old and have been passionate about computer science since high school. After graduating from a STI2D baccalaureate with a specialization in digital systems, I joined the Bachelor of Technology in Computer Science at the IUT of Bayonne. In my third year, I had the opportunity to enroll in a double degree program at the Université du Québec à Chicoutimi (UQAC).

My internship took place at UQAC within the computer science department. Supervised by Mr. Kevin Bouchard, the project focused on developing a Python library called PyAdverseSearch, dedicated to adversarial search algorithms, often used in zero-sum games. I contributed to designing the architecture, developing core components (such as the Game, State, Node classes), and implementing Minimax and Monte Carlo Tree Search algorithms.

Teamwork played a key role, with each member assigned a specific role: speaker, destroyer, clean coder, etc. I was the "oracle," in charge of strategic planning and weekly coordination.

This internship helped me deepen my skills in algorithms, object-oriented programming, as well as project management and collaboration. It also gave me insight into the differences between French and Canadian working methods and reinforced the importance of optimization and pedagogy in technical environments.

I would like to sincerely thank Mr. Kevin Bouchard for his guidance, and all team members for their involvement in this ambitious project.