Rapport final MAP08

PSC : **“Analyse du jeu de Dots-and-Boxes”**
Code PSC : **MAP08**
Coordinateur : Teddy Pichard
Tuteur : Lucas Gerin
Membres : BENYAMINE Axel, BORDERIES Titouan, COLLOMB Louis, DELPECH de SAINT GUILHEM Dimitri, DIBY Wilfried.

Ce PSC s'intéresse au jeu Dots-and-Boxes, à travers une analyse théorique et expérimentale du jeu.

Description du jeu.

Le jeu se joue sur une grille de points. Chacun leur tour, les joueurs relient verticalement ou horizontalement deux points voisins par un trait. Lorsqu’un joueur ferme un carré, il le marque, gagne un point et rejoue. Le jeu se termine quand tous les traits ont été tracés. Le joueur qui a fermé le plus de carrés gagne.

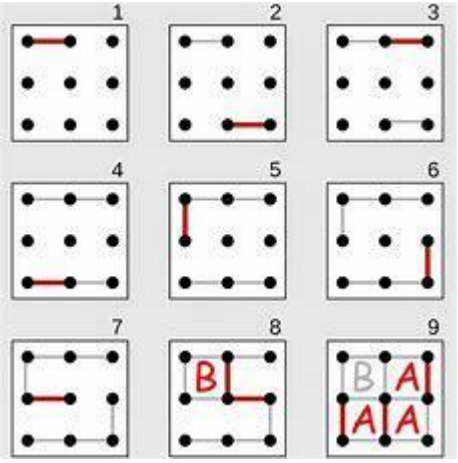

Figure 1 : Le joueur A gagne la partie. Source : [7] (cf. bibliographie)

Enjeux et motivations du projet.

Le jeu Dots and Boxes est un jeu tout à fait intéressant et pertinent à étudier. Tout d'abord, le nombre de configurations possibles dans le jeu est très élevé. Pour une grille de taille $m \times n$ points, il y a $(m-1)n + (n-1)m$ lignes possibles, ce qui donne un grand nombre de positions possibles. Le nombre de positions atteint rapidement des valeurs astronomiques pour les tailles de grille supérieures à 5x5, ce qui rend l'étude exhaustive des positions impossible même pour les ordinateurs les plus puissants.

Ensuite, la difficulté du jeu dépend largement de la stratégie utilisée par l'adversaire. Le choix optimal à chaque tour dépend de la stratégie de l'adversaire, ce qui rend difficile la planification d'une stratégie gagnante à long terme. Il peut y avoir plusieurs équilibres de Nash, ce qui signifie qu'il peut y avoir plusieurs séquences de coups optimales qui mènent à une victoire.

De plus, l'évaluation de la position actuelle peut être complexe. Les algorithmes d'évaluation doivent prendre en compte des facteurs tels que le nombre de carrés remplis par chaque joueur, le nombre de carrés potentiellement remplissables, la structure de la grille et les positions défensives et offensives.

Enfin, le jeu est très sensible aux erreurs d'approximation dans les algorithmes d'évaluation. Une légère erreur dans l'évaluation d'une position peut mener à une mauvaise prise de décision qui peut coûter la partie.

En résumé, Dots and Boxes est un jeu complexe à étudier d'un point de vue informatique en raison de la grande variété de configurations possibles, de la dépendance de la stratégie de l'adversaire, de l'évaluation complexe des positions et de la sensibilité aux erreurs d'approximation. Cependant, malgré ces difficultés, les chercheurs continuent de développer des algorithmes performants pour jouer au jeu de manière efficace et optimale.

Nous allons essayer de déterminer une stratégie gagnante pour des grilles de tailles supérieures à 5 x 4. Une grille de 5 x 4 étant la grille de taille maximale pour laquelle un algorithme naïf a été capable de résoudre le jeu. Nous chercherons aussi à comprendre les concepts phares du jeu Dots-and-Boxes. Le point clé de ce PSC est la création d'un algorithme solveur.

Pour faire cela, nous voulons analyser différents algorithmes existants, et comprendre leurs structures, leurs avantages ainsi que leurs inconvénients. Nous pourrons ainsi répondre à certaines de nos questions : Quels sont les algorithmes les plus performants ? Comment peut-on potentiellement les améliorer ? Sont-ils adaptables ou applicables à des grilles plus originales ?

Nous nous intéresserons en particulier aux algorithmes : *Monte Carlo Tree Search* et *MiniMax* afin d'élaborer et raffiner des stratégies intuitives. La résolution théorique repose sur l'utilisation du concept de *longues chaînes*.

Sommaire

I. Bases théoriques

a) Notations et terminologie

La littérature associée étant principalement en anglais, il a été décidé de conserver en anglais certaines notations clefs qu'on utilisera pour la suite.

Dans toute la suite, on nommera Joueur A et joueur B respectivement le joueur qui débute la partie et celui qui joue en second.
La règle suivante est cruciale: quand une boîte peut être potentiellement fermée par un des deux joueurs, il n'a pas l'obligation de le faire. Cette règle en apparence sans importance est à l'origine de toute la complexité de ce jeu.

On définit la **valence** d'une boîte comme étant le nombre de lignes qu'il reste encore à tracer pour remporter la boîte. Ainsi la valence est un entier positif compris entre 0 et 4.

Une **chaîne** est alors un groupe de boîtes voisines de valence deux et une **longue chaîne** (*long chain* en anglais) peut être définie de la manière suivante : une chaîne d'au moins trois boîtes.

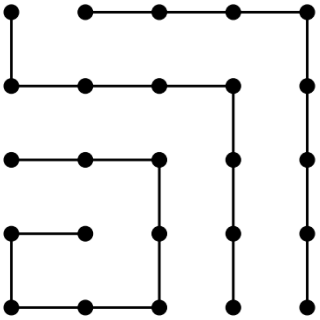

Figure 2 : Des exemples de longue chaîne. Source : [6]

Un joueur est dit en **contrôle de la partie** lorsqu'il force son adversaire à faire passer la valence d'une des boîtes d'une longue chaîne de deux à un.
Lorsqu'un joueur complète deux cases avec un seul coup on parle de ***doublecrossed moves***. Nous l'appellerons coup double en français.
On appelle ***double-dealing*** le fait de refuser de prendre les deux dernières boîtes d'une longue chaîne après l'avoir complété en partie. Cette technique permet de garder le contrôle de la partie. En effet, le joueur qui complète une longue chaîne sauf deux cases évite ainsi d'amorcer une autre longue chaîne pour son

adversaire en créant deux boîtes de valence un. Il crée une situation de coup double pour son adversaire.

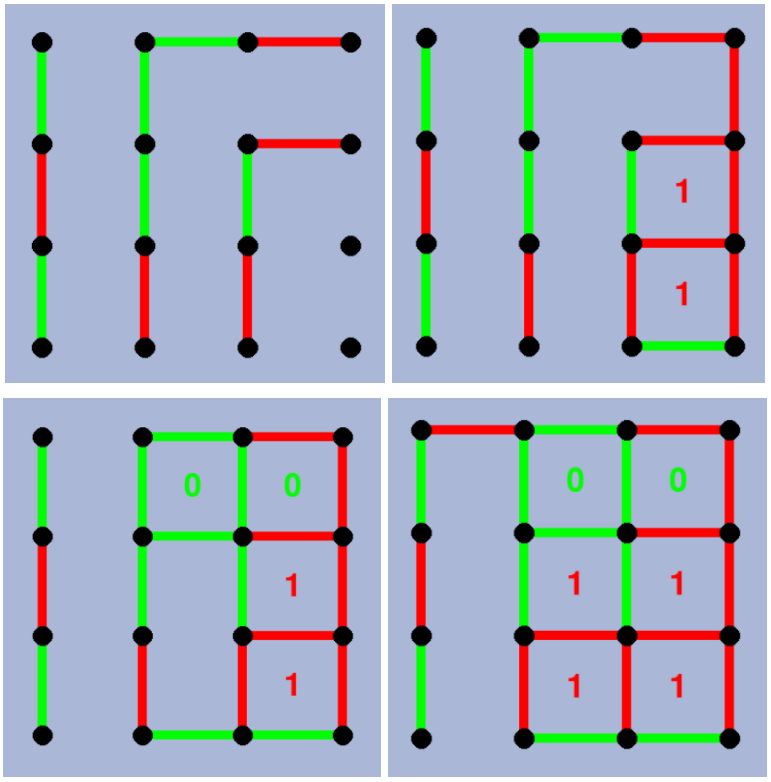

Figure 3 : Coup double-dealing du joueur A en 3ème image.

Il y a également des **boucles.** Ce sont des blocs qui sont entourés, on peut les voir comme des chaînes qui se terminent sur elles-mêmes. Enfin, la dernière forme de chaîne sont les chaînes en **branches.** C'est une chaîne qui se divise en deux.

Dots-and-Boxes est un jeu dit *impartial* en ce que les coups disponibles dépendent seulement de la configuration de la grille et non de quel est le joueur actuel (à la différence du jeu d'échecs par exemple où chaque joueur a ses propres pièces).

b) Quelques résultats

Structure du problème, première approche:

Les règles du jeu Dots-and-Boxes sont relativement simples, mais l'espace des grilles est très grand. En effet, un jeu de taille $m \times n$ points a $p = m(n-1) + (m-1)n$ coups possibles et donc 2^p grilles différentes.
Une première exploration naïve va donc générer $p!$ grilles, le problème devient donc vite insoluble. Cela montre la nécessité de trouver un solveur efficace.

Règle des *long chains* [1]

Le joueur A (respectivement joueur B) a intérêt à ce que la somme du nombre de coup double et du nombre de points soit impaire (respectivement paire).

Cette formulation est équivalente à la suivante que nous utiliserons plus tard: Le joueur A (respectivement joueur B) a intérêt à ce que la somme du nombre de *long chains* et du nombre de points soit paire (respectivement impaire).

Chaque coup qui complète une boîte ajoute un coup pour le tour en cours à l'exception des coups doubles. On obtient la relation suivante :

Nb de tours complétés = Nb de coups + Nb de coups doubles - Nb de carrés

Comme il y a $m \times n$ points donc $m(n-1) + n(m-1)$ coups à jouer (horizontaux et verticaux) et $(m-1)(n-1)$ carrés, on obtient

Nb de tours complétés = $mn - 1$ **+ Nb de coups doubles**

Le dernier tour est incomplet car un carré est complété mais le joueur ne rejoue pas donc on obtient :

Nb de tour = Nb de points + Nb de coups doubles

Le joueur qui complète la dernière chaîne ne réalise pas de *double-dealing* car il n'a plus besoin de garder le contrôle. Ainsi nous pouvons raisonnablement dire que le joueur qui termine la partie la gagne puisqu'il était en contrôle de la partie et que, sauf erreur, il l'avait depuis le début des points marqués.

Le joueur A jouant toujours les tours impaires, afin de gagner il doit faire en sorte que le nombre de tour est impair. Puisque un *double dealing* n'est pas fait

lors de la complétion de la dernière chaîne, le nombre de *long chain* est de parité différente de celui du nombre de coup double.
Lorsque l'on compte les *long chains*, il faut compter également le nombre d'embranchement. En effet, si l'on commence à compléter le début d'une chaîne qui termine sur un embranchement, il faut faire un double dealing pour fermer la chaîne au niveau de l'embranchement et compter ce qu'il reste comme une chaîne.

Chaînes de deux cases [1]

Les situations rencontrées avec des chaînes de deux carrés peuvent être rangées en deux catégories. Les ***hard-hearted handout*** et les ***half-hearted handout*** comme les nomme Berkelamp.
La première catégorie regroupe les situations dans lesquelles le contrôle de la partie est préservé. C'est dans ces situations que tout joueur en contrôle souhaite mettre son adversaire. Le *double-dealing* en fin de chaîne en est un cas particulier. Dans ces situations l'adversaire est forcé de prendre les deux carrés et de rejouer.

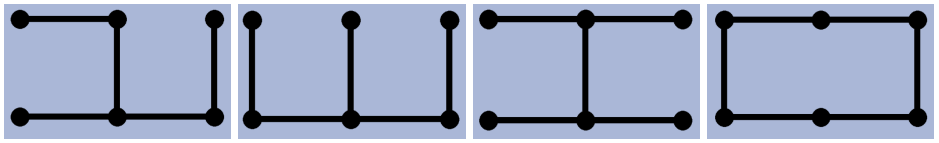

Figure 4: Situations de hard-hearted handout.

La deuxième laisse le choix à l'adversaire de prendre les carrés ou de réaliser un *double dealing*. Il faut éviter cette situation car l'adversaire peut reprendre le contrôle. Cependant si vous n'êtes pas en contrôle, c'est l'occasion de le prendre sur une erreur d'inattention de l'adversaire.

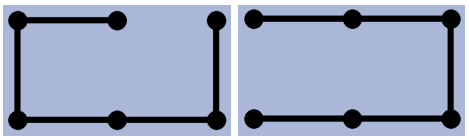

Figure 5: Situations de half-hearted handout.

II. Objectifs initiaux

1. **Implémenter des solveurs de "dots-and-Boxes" suivant différentes méthodes : IA, backtracking, Monte-Carlo tree search, ou autre…**

Ces algorithmes sont des algorithmes standardisés très utilisés dans la résolution de jeux mathématiques.
Dans un premier temps, il convient de comprendre le fonctionnement théorique de ces algorithmes et d'en identifier certaines caractéristiques (déterminisme ou non, complexité, …). Dans un second temps, il s'agira d'adapter ces algorithmes théoriques aux spécificités de "Dots-and-Boxes" puis de les implémenter.

2. **Comparer les différents solveurs sur des grilles standard**

La comparaison se fera selon plusieurs critères : temps d'exécution, validité de la solution proposée par le solveur, données nécessaires en amont, le solveur joue-t-il comme un humain ? etc.

3. **Tenter de concevoir un solveur innovant**

Deux méthodes sont envisageables pour concevoir un tel solveur:

- Établir de nouveaux théorèmes qui permettent de gagner plus facilement
- Établir une fonction score innovante (fonction qui à une grille donnée renvoie un score entre 0 et 1 pour chacun des deux joueurs, ce score représente la tendance de chaque joueur à gagner la partie à partir de cette position) et s'en servir.

4. **Analyser théoriquement le jeu sur des grilles moins standards (grille triangulaire, pavage régulier, grilles 3D…)**

Certaines propriétés (comme par exemple celle des *long chains* ou du *double-dealing*) restent-elles valables ?
Nous commencerons par nous approprier davantage les résultats existants et leur preuve, pour ensuite chercher à les étendre et les adapter à des grilles différentes.

5. **Analyser théoriquement ou expérimentalement le jeu sur des grilles aléatoires.**

Il convient dans un premier temps de générer de telles grilles, à travers la génération de polyèdres avec un nombre d'arêtes par face fixé. La résolution s'appuiera ensuite sur les travaux précédents.

III. Implémentation du jeu

Nous avons réalisé une interface graphique pour le jeu de Dots-and-Boxes en Python. Celle-ci nous permet de réaliser une partie à plusieurs joueurs, ceux-ci étant humains ou non. Nous pouvons également réaliser la partie sur une grille à motif carré mais qui n'est pas rectangulaire. Elle reste cependant connexe.

a) Programmation d'une partie

Nous avons modélisé le jeu en python en créant une classe jeu. Il contient notamment 3 matrices. Une décrivant les barres verticales, une pour les barres horizontales et une contenant les valences des carrés. Pour chacun des traits dans les deux premières matrices, il y a un booléen indiquant si la barre est tracée ou non. La dernière matrice garde la valence des carrés. L'objet a alors une méthode pour vérifier la valence des carrés, pour jouer un tour, donner le score et afficher la grille. Par ailleurs chaque joueur a une variable score qui est incrémenté au fur et à mesure.
Lorsque l'utilisateur clique, l'algorithme cherche l'arête la plus proche et trace un trait. L'utilisateur peut ainsi cliquer dans le carré de diagonale l'arête concernée.
Quand un carré est complété, le numéro du joueur est écrit dans sa couleur au centre du carré. En fin de partie, le jeu efface la grille et relance une partie.

Chaque coup est représenté par un triplet formé de deux entiers et un booléen. Les deux entiers indiquent le point de départ et le booléen indique la direction du trait, True pour vertical et False pour horizontal.

Le score du jeu est affiché sur le côté de l'écran dans l'ordre de jeu des joueurs. On utilise la librairie PyGame qui nous permet de tracer des formes géométriques sur l'interface. L'interface est ensuite actualisée et retrace toute la grille à chaque changement du jeu.

b) Programmation pour plusieurs joueurs et algorithmes

Notre interface de jeu permet d'afficher une grille de jeu que cela soit pour un nombre quelconque de joueurs humains, un ordinateur et un humain ou encore deux algorithmes.

Pour faire cela nous fournissons une liste d'objets "joueur". Cette liste est de longueur quelconque.

Chaque joueur possède un numéro; une couleur de représentation sur la grille; un score, mis à jour au fur et à mesure que la partie avance; un booléen indiquant le caractère humain du joueur; l'algorithme associé dans le cas d'un joueur non humain.

La fonction d'affichage de score s'adapte pour afficher plusieurs nombres. Lorsqu'il y avait deux joueurs, le contenu du tableau des carrés indiquant quel joueur l'avait complété. Ce n'est plus le cas maintenant, le score est mis-à-jour à chaque carré complété.

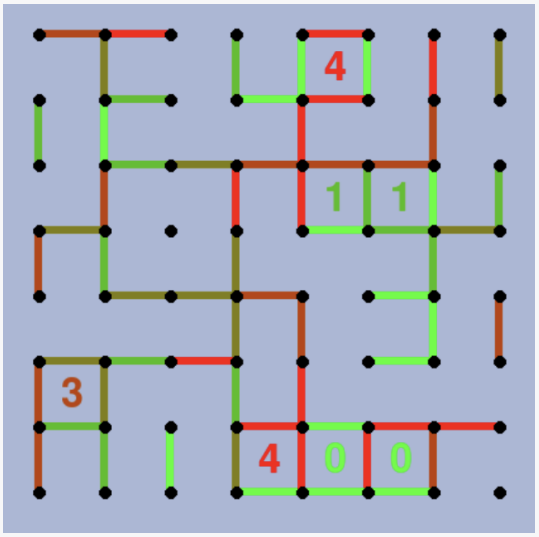

Figure 6: Exemple de partie à 5 joueurs.

c) Génération de grille non rectangulaire au motif carré

Initialement, les variantes que l'on souhaitait implémentées étaient celles de grilles triangulaires, hexagonales ou de quadrilatère. Cependant la structure du code et l'implémentation graphique n'étaient adaptées qu'aux grilles à motif carré. Puisque nous avions déjà rencontré des difficultés avec le code pour cette première implémentation, nous avons conservé seulement le motif carré.

Deux méthodes ont été envisagées pour générer ces grilles : une approche plutôt générative en ajoutant des carrés et une approche destructive en retirant à chaque fois des carrés d'une grille plus grande. C'est la première approche qui a été retenue.

Directement dans la classe Jeu, nous avons implémenté une méthode permettant de générer une grille non rectangulaire au motif carré. Nous considérons tout de même ces grilles comme des sous-grilles d'une grille rectangulaire plus grande, un peu comme dans la deuxième approche. On procède alors de la manière suivante pour générer une grille :

- **Initialement** : choisir un carré aléatoirement.
- **Tant que** l'on n'a pas la taille souhaitée :
    - choisir uniformément un carré qui admet un voisin non sélectionné
    - choisir uniformément un de ses voisins parmi ceux non sélectionnés
    - L'ajouter et mettre à jour l'ensemble des carrés qui peuvent être choisis.

Cela garantit la connexité de la grille obtenue.

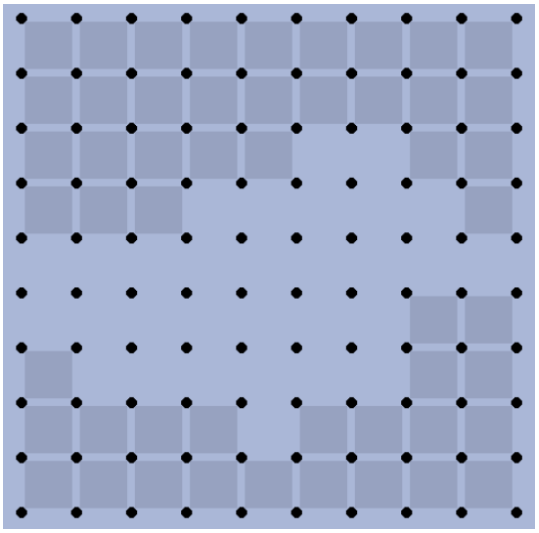

Figure 7: Grille de 30 carrés dans une grille 9x9.

d) Affrontement à grande échelle

Nous utilisons les points au-dessus pour implémenter une fonction qui permet de réaliser des affrontements entre les différents solveurs que nous avons implémentés. Ceux-ci s'affrontent sur un nombre choisi de parties afin de déterminer les proportions de victoires.
Ces affrontements se réalisent sur des parties à 2 ou 3 joueurs et sur des grilles rectangulaires ou non. Nous interpréterons les résultats plus loin.

e) Structure globale

Tout d'abord, les différents solveurs suivent la même structure de code globale. Chaque solveur est une classe qui hérite de la classe abstraite `Policy`. Celle-ci ne contient qu'un constructeur et une fonction `select_edge` qui prend au moins en argument la taille de la grille ainsi que son état. C'est elle qui à un état du jeu retourne le coup à jouer.

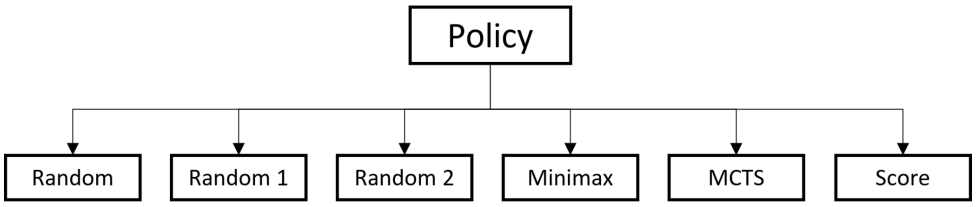

Figure 8: Ensemble des policy considérés

Random, Random 1, Random 2, Minimax, MCTS et Score sont les différents agents que nous avons implémentés. Du fait de la répartition des tâches, la représentation des arêtes est différente pour les agents et pour l'objet `Jeu`. En effet, celui-ci emploie des matrices plus adaptées à la représentation graphique tandis que les agents considèrent l'ensemble des coups comme une liste. Nous avons donc dû coder des fonctions de conversion.

Dans un autre fichier, nous avons codé toutes les fonctions nécessaires à l'affichage. Celui est nommé `front_end`. Le schéma suivant donne l'articulation des différents fichiers.

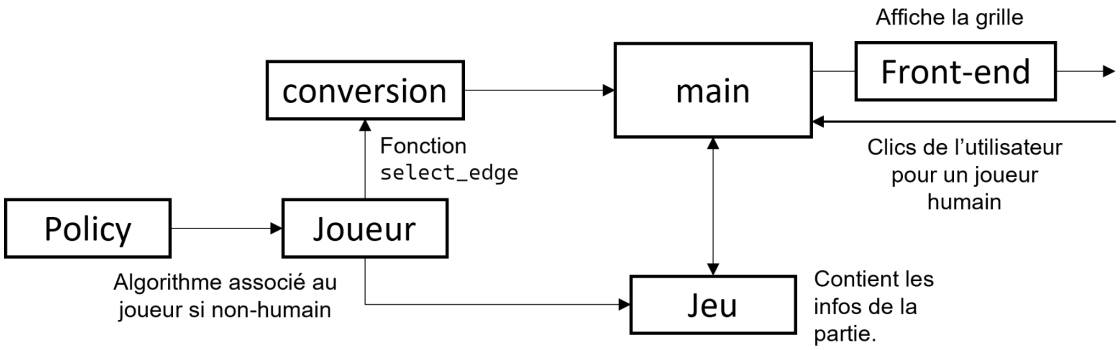

Figure 9: Diagramme de l'implémentation du jeu et des policy

IV. Les différents solveurs

Nous voulions disposer a priori de plusieurs joueurs de niveaux distincts dans la perspective de pouvoir comparer leurs performances. Ainsi nous avons tout d'abord implémenté l'algorithme le plus simple auquel on peut penser.

a) Différents solveurs aléatoires

Le premier solveur, “Random” choisit aléatoirement parmi tous les coups légaux. Cet algorithme nous servira de référence dans l’évaluation des autres agents. Notamment dans l’évaluation temporelle des autres algorithmes. Puisqu’il sélectionne le coup à jouer en temps constant, le temps qu’il indiquera sera en grande partie celui nécessaire au déroulement des parties avec notre implémentation.

Le solveur “Random 1” suit la même politique de décision que Random à l’exception près que dès qu’il en a l’occasion il complète directement une boîte.

Le solveur “Random 2” suit également la même politique de sélection que Random 1 mais il ne sélectionne pas des coups qui permettraient à l’adversaire de compléter directement une boîte. Ces trois algorithmes sont les plus simples auxquels on peut penser et permettent de facilement et rapidement interpréter les résultats obtenus pour les autres agents.

Ces solveurs n’ont pas posé de difficulté particulière à être implémentés. Cependant, nous avons réfléchi à la manière la plus pertinente de structurer l’architecture globale du code, notamment sur comment coder intelligemment l’état du jeu. Nous avons pour notre part choisi de numéroter dans un ordre arbitraire les N arêtes du jeu et de créer une liste binaire de taille N, représentant les deux possibilités respectives pour chaque arête: tracée ou non.

b) Minimax

L’algorithme le plus intuitif auquel on peut ensuite penser pour résoudre le jeu de Dots and Boxes est appelé ***Minimax***. C’est un algorithme récursif et déterministe dans le sens où il donnera toujours le même résultat pour une même entrée. L’algorithme Minimax est un algorithme utilisé pour les jeux à deux joueurs avec des règles définies et un état final déterminé. Il consiste à maximiser les gains pour un joueur 1 tout en minimisant les gains pour l’autre joueur 2 . L’algorithme parcourt l’arbre des possibilités de la partie et utilise une fonction de notation pour évaluer un état terminal possible d’une partie, et en choisissant l’action qui mène à l’état le plus avantageux pour le joueur 1. Il s’agit d’une stratégie radicale qui maximise le gain d’un joueur en partant du principe que son adversaire joue le meilleur coup possible.

La grande faiblesse de l’algorithme Minimax est la nécessité de développer l’entièreté de l’arbre des possibilités. Pour des jeux dont l’arbre des possibilités contient un nombre de branches très important (c’est le cas de notre jeu à partir d’une certaine dimension de la grille), on est assez limité computationnellement. En effet, le Minimax possède une très mauvaise complexité temporelle, par

exemple si l'on considère un plateau de jeu de taille 3X3, il y a déjà 18! états de jeu à évaluer. Pour résoudre le problème, on développe l'arbre des possibilités jusqu'à une certaine profondeur puis on évalue les états du jeu grâce à une fonction qui donne un score aux nœuds de l'arbre qui correspondent aux différents états possibles. On implémente classiquement l'algorithme MiniMax grâce au pseudocode suivant:

```
function Minimax(noeud, profondeur, maximizing)
        if noeud is feuille then
                return evaluation(noeud)
        else if maximizing is true then
                score  ← -infini
                for each enfant of noeud
                        score ← max(score, Minimax(enfant, profondeur -1, false))
                return score
        else if maximizing is false then
                score ← +infini
                for each enfant of noeud
                        score ← min(score, Minimax(enfant, profondeur -1, true))
                return score
end Minimax
```

La fonction Minimax prend en entrée trois argument, le noeud qui représente un état du jeu, la profondeur de recherche (qui garde en mémoire le nombre de coups joués jusqu'à présent pour atteindre le noeud actuel) et la variable booléenne maximizing qui permet d'avoir accès au joueur dont c'est le tour de jouer. Si c'est true, cela signifie que c'est au tour du maximizer, et dans ce cas, on garde le score le plus élevé de tous les enfants du nœud actuel. On répète ce processus jusqu'à atteindre une feuille de l'arbre. La fonction évaluation donne en sortie le résultat d'une certaine fonction d'évaluation d'un état de jeu.

Le problème de Minimax est que les informations ne circulent que dans un seul sens : des feuilles vers la racine. Il est ainsi nécessaire d'avoir développé chaque feuille de l'arbre de recherche pour pouvoir propager les informations sur les scores des feuilles vers la racine.

Le principe de l'***AlphaBeta prunning*** (élagage AlphaBeta), que nous appellerons par abus de langage algorithme AlphaBeta, est précisément d'éviter la génération de feuilles et de parties de l'arbre qui sont inutiles. Pour ce faire, cet algorithme repose sur l'idée de la génération de l'arbre selon un processus dit en « profondeur d'abord » où, avant de développer un frère d'un nœud (rappel : deux nœuds frères sont des nœuds qui ont le même parent), les fils sont développés. A

cette idée vient se greffer la stratégie qui consiste à utiliser l'information en la remontant des feuilles et également en la redescendant vers d'autres feuilles.

Plus précisément, le principe de AlphaBeta est de tenir à jour deux variables α et β qui contiennent respectivement à chaque moment du développement de l'arbre la valeur minimale que le joueur peut espérer obtenir pour le coup à jouer étant donné la position où il se trouve et la valeur maximale. Certains développements de l'arbre sont arrêtés car ils indiquent qu'un des joueurs a l'opportunité de faire des coups qui violent le fait que α est obligatoirement la note la plus basse que le joueur 1 sait pouvoir obtenir ou que β est la valeur maximale que le joueur 2 autorisera 1 à obtenir.

c) Monte Carlo Tree Search

Après avoir implémenté des solveurs naïfs, la prochaine étape consistait à implémenter des solveurs avec des algorithmes un peu plus élaborés.

L'algorithme Monte Carlo Tree Search (MCTS) est une technique d'exploration de l'arbre de recherche utilisée pour résoudre des problèmes de décision dans des environnements incertains. Il est souvent utilisé pour résoudre des jeux à deux joueurs tels que les échecs, le Go ou le poker.

L'algorithme Monte Carlo tree search constitue une méthode classique pour les problèmes avec un espace de recherche trop grand pour être exploré avec un temps de calcul raisonnable par l'algorithme Minimax. Un avantage de ***MCTS*** est qu'il ne nécessite pas de fonction de score a priori pour être efficace. Or l'on connaît la difficulté de construire une fonction de score pertinente pour le jeu dots and boxes. La seule information nécessaire est comment l'état du jeu change quand une action quelconque est effectuée.
Aucune stratégie heuristique n'a également besoin d'être implémentée. En effet, le parcours de l'arbre (tree search) n'a pas besoin d'évaluer des actions intermédiaires. Les simulations sont jouées jusqu'à la fin de la partie (jusqu'aux feuilles de l'arbre). Ainsi, l'algorithme doit simplement être capable de déterminer le gagnant final (en comptant le nombre de boîtes complétées par chaque joueur dans le cas de Dots and Boxes).

Chaque nœud de l'arbre contient trois informations (chaque nœud représente une configuration de jeu). Tout d'abord le nombre de fois que le nœud a été parcouru par l'algorithme, on note ce nombre $N(s)$ avec s l'état du jeu (state), puis le nombre de fois qu'une action a été choisi en partant de cet noeud, on le note $N(s,a)$, enfin la récompense totale pour toutes les actions de toutes les simulations dans lesquelles cette action a été choisie $W(s,a)$. La récompense pour

une seule simulation peut être binaire ou non. Dans le cas de Dots and Boxes, il semble plus pertinent de procéder de manière binaire étant donné que seule la victoire importe quel que soit le nombre de boîtes complétées. Autrement dit, on ne prend pas en compte la marge de victoire. La récompense moyenne d'une action a dans un état de jeu *s* est calculée comme ceci $Q(s,a) = W(s,a)/N(s,a)$.

Une simulation de ***MCTS*** possède quatres étapes distinctes, la sélection, l'expansion, le playout et la backpropagation.
Pendant la phase de ***sélection***, l'arbre existant est traversé en utilisant une certaine politique de sélection à chaque nœud *s*. Parmi toutes les politiques de sélection qui existent, la plus intuitive et celle la plus communémment utilisée est de sélectionner l'action avec la plus grande récompense moyenne (ie $a = max_a Q(s,a)$).
La phase d'***expansion*** est la phase où de nouveaux nœuds jusqu'alors inexplorés sont ajoutés à l'arbre. On peut choisir d'ajouter à l'arbre tous les nouveaux nœuds, ou d'exiger un certain nombre arbitraire $N(s,a)$ à atteindre pour être ajouté. Après l'exploration, la phase de ***playout*** commence.
Pendant cette phase cruciale, la simulation continue jusqu'à atteindre une feuille de l'arbre. Différentes politiques de sélection peuvent être implémentées, soit une sélection complètement aléatoire (***random policy***) ou bien une politique basée sur une stratégie heuristique. Finalement, à la fin de la partie, les valeurs des différents nœuds s sont actualisées selon le résultat de la partie. On appelle cette phase la ***backpropagation***.
Plusieurs approches sont pertinentes pour cette phase, on peut attribuer à chaque simulation le même poids ou bien on peut pondérer l'actualisation en prenant davantage en considération les dernières simulations puisqu'elles disposaient de plus d'informations pour sélectionner des actions efficaces.

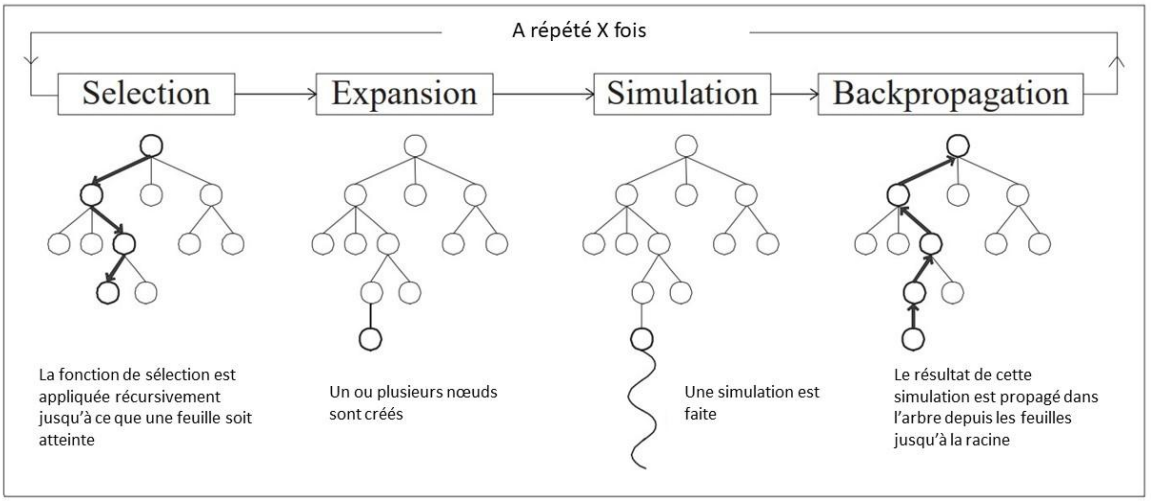

*Figure 10 : Représentation des quatres phases de l'algorithme **MCTS**. Source : [3]*

Ainsi, le ***MCTS*** s'implémente classiquement de la manière suivante. Dans cet algorithme, on désigne par *S* l'ensemble des noeuds de l'arbre de recherche (c'est à dire l'ensemble des états de jeu accessible depuis la racine), *Selection(Noeud N)* est la fonction de sélection, qui donne en sortie un enfant du noeud N. *Expansion(Noeud N*) est la fonction qui ajoute un nouveau noeud à l'arbre et retourne ce noeud. *Jouer_simulation(Noeud N)* est une fonction qui simule une partie de jeu en partant du nœud N et retourne le résultat R (-1, 0 ou 1) en fonction du gagnant de la partie. *Rétropropagation(Int R)* est une procédure qui actualise la valeur du nœud qui dépend du résultat R de la dernière partie simulée. N_e *(noeud N)* est l'ensemble des enfants du noeud N.

```
while(reste_temps)

    noeud_actuel noeud_racine
    while(noeud_actuel ∈S)

        dernier_noeud ⟵ noeud_actuel
        noeud actuel ⟵ Selection(noeud_actuel)    #Sélection

    dernier_noeud  ⟵ Expansion(dernier_noeud)          #Expansion
    R ⟵  Jouer_simulation(dernier_noeud)              #Simulation
    while(noeud_actuel ∈ S)

        noeud_actuel.Retropropagation(R)          #Backpropagation
        noeud_actuel.compter_visite  noeud_actuel.compter_visite +1
        noeud_actuel.parent

return meilleur_coup = argmax(N.compter_visit), pour N ∈ N_e(noeud_racine)
```

Upper confidence bound for trees:

Ainsi, il est clair que l'algorithme ***MCTS*** utilise les informations de précédentes simulations pour guider les simulations futures. Cependant, cela ne permet pas de découvrir de potentielles meilleures stratégies. On risque en effet de n'aboutir qu'à un minimum de potentiel local. Un enjeu majeur de ***MCTS*** est donc de trouver un compromis entre l'exploitation des précédentes simulations (c'est à dire utiliser les valeurs et les noeuds de l'arbre déjà exploré pour affiner la politique de choix) avec l'exploration (biaiser la politique dans l'objectif de découvrir des branches inexplorées).

Une façon classique de le faire est d'utiliser un upper confidence bound, qui est une estimation de la récompense maximale réelle, étant donnée la confiance relative en la valeur $Q(s,a)$. Autrement dit, au lieu de systématiquement choisir

l'action a à partir de s qui possède le meilleur résultat estimé $Q(s,a)$, on ajoute un terme d'exploration.
On se rend rapidement compte que plus $N(s)$ et $N(s,a)$ sont grands, plus $Q(s,a)$ est une estimation pertinente de la valeur réelle de l'action. La bonne expression est donnée par $Q^*(s,a) = Q(s,a) + \sqrt{log(N(s) \div N(s,a)}$.

d) Solveur basé sur une fonction de score pertinente

L'idée d'un solveur basé sur une fonction de score est d'implémenter un solveur qui prendrait en considération des résultats particuliers sur le jeu Dots and Boxes. Pour cela il faut d'abord définir une fonction score qui a une grille donnée renvoie un score pour chacun des joueurs. Ce score symbolise alors l'avantage qu'aurait le joueur en question face à cette grille.

Afin de produire cette fonction score, il faut en premier lieu tenter d'apprendre à jouer le mieux possible en tant qu'humain via des théorèmes mathématiques. Une fois cette fonction de score définie, il reste alors à l'appliquer dans le cadre d'un solveur afin de le rendre plus efficace et de tenter de le faire jouer davantage comme un humain le ferait.

Étant donné la difficulté du jeu, nous nous sommes intéressés aux résultats d'Elwyn Berlekamp dans son ouvrage *The dots and Boxes game (A K Peters)* [1]. Il y explique des techniques, plus ou moins simples d'application, pour mieux jouer à dots and boxes.

La technique la plus simple consiste à compter les *long chains* (cf. terminologie). Les informations récoltées (la parité du nombre de *long chain* et leur longueur) permettent d'identifier l'avantage du joueur A ou du joueur B en supposant que les deux joueurs jouent parfaitement.

La **règle des *long chains*** (voir section Quelques résultats) permet de connaître le joueur ayant l'avantage. Cet avantage se quantifie alors en fonction de la longueur des *long chains* de la grille (en effet, si toutes les *long chains* sont de longueur 3, le joueur ayant l'avantage devrait seulement fermer 3 cases de plus que son adversaire).
La règle des *long chains* prend également en compte le cas du double-dealing puisque l'action qui consiste à laisser les deux cases change le joueur sans changer la parité du nombre de *long chain*.

D’autres résultats, nettement plus complexes à implémenter, existent mais nécessitent des temps de calcul bien plus longs. Nous nous sommes donc concentrés sur ce résultat précis.

Implémentation

Afin de choisir l'arête à jouer, notre fonction score a en entrée l’identifiant du joueur qui doit jouer le coup, le nombre de carrés fermés par chacun des joueurs ainsi que le nombre de *long chain* et leurs longueurs.

Tout d’abord, nous sommes parvenus à implémenter un détecteur de *long chain*. Le code de cette fonction constitue en un appel à la fonction récursive `prochain_chaine` dont le principe général est écrit ci-dessous.

Dans le code suivant, chaine est une matrice (initialement vide) dont chaque case correspond à un carré de la grille. Une direction désigne haut, bas, gauche ou droite. Un voisin désigne un carré adjacent où l'arête commune entre le carré et le carré adjacent n’a pas été placée.

```
fonction prochain_chaine(carré, direction_origine=nulle):
      Si le carré a une valence de 2 :
      On parcourt les voisins possibles et on s'arrête lorsque la direction
carré->voisin est différente de direction_origine (Si direction_origine est
nulle on prend le premier voisin)
      On note voisin le voisin retenu et dir la direction carré->voisin
      On note opposée la direction opposée à dir
      chaine[carré] = dir
      Si voisin n’a pas déjà été  parcouru (ie. chaîne[voisin] est vide):
            prochain_chaine(voisin, opposée)
```

Une fois les appels effectués, on obtient une matrice “chaîne” qui permet de suivre le parcours de toutes les chaînes de la grille. On obtient ainsi de nombreuses informations dont le nombre de *long chains* et leur longueur respective.

La seconde étape de la conception de cette fonction score est l’équilibrage entre l’importance donnée au score actuel de la configuration (nombre de carrés fermés par chaque joueur, le nombre de *long chain* et leur longueur).

Supposons que l’on s’intéresse au joueur A.
L’idée est de concevoir une fonction affine dont l’équation est score = $ax + b$ où:

- b est le nombre de carré fermés par A - nombre de carrés fermés par B

- *a* est positif si la “somme du nombre de points initiaux (m x n) et du nombre de *long chains* est pair” et négatif sinon (cf. III.3.a)). Sa valeur absolue est choisie en fonction de l’importance relative des *long chains* par rapport aux carrés déjà fermés par le joueur.
- *x* est la longueur moyenne des *long chains*.

Le score pour le joueur B est alors l’opposé du score du joueur A (car *a* est l’indicatrice associée à l’imparité).

Initialement, nous avions l’idée de prendre le min des longueur pour x car si a est positif, A aura une *long chain* de plus que B s’il joue parfaitement. Mais si B joue parfaitement, cette *long chain* sera la plus petite. En réalité, l’impact des *long chains* est trop faible si l’on considère uniquement le minimum. La moyenne de la longueur des *long chains* caractérise mieux la grille et permet d’avoir une influence plus importante sur le score.

Ici, un score positif indique un avantage pour le joueur considéré, plus la valeur est élevée en valeur absolue et plus l’avantage est prononcé.
Cette fonction score constitue un exemple simple d’une fonction de score utilisant la règle des *long chain* (III.3) mais n’est probablement pas la meilleure.

Enfin, l’algorithme basé sur cette fonction a deux profondeurs possibles : 1 ou 2. Dans le cas de la profondeur 1, l’algorithme remplit une arête, calcule le score de la grille obtenue puis enlève l’arrête. Il répète l’opération pour toutes les arêtes et renvoie celle ayant obtenu le meilleur score.
Dans le cas de la profondeur 2, l’algorithme remplit une arête puis une autre (coup simulé du joueur suivant), calcule le score de la grille obtenue puis enlève les deux arêtes. Il répète alors l’opération pour tous les couples d’arêtes possible et renvoie la première arête du couple ayant obtenu le meilleur score.

Contraintes et choix à faire vis à vis de l’implémentation

Au sujet du détecteur de *long chain*, la difficulté est de mettre à jour la matrice "chaîne" (du fait de la récursivité et de certains cas particuliers comme des fusions de *long chain*). Ainsi, nous re-calculons la matrice “chaîne” dès qu’il y en a besoin, en repartant de zéro ce qui implique un temps de calcul notable.

Le choix de |a| permet de quantifier l’importance relative des *long chains* par rapport aux carrés déjà fermés par le joueur.
Nous avons étudié son influence de manière très classique, en parcourant les différentes valeurs de a possible et en notant les taux de victoire associés pour

différentes tailles de grille. Nous avons trouvé deux cas limites intéressants (a=1 et a=100). Bien souvent (selon n et p) l'un de ces deux cas limite est un optimal.

e) Apprentissage machine et Dots and Boxes

A travers nos recherches bibliographiques, nous nous sommes rendus compte que les solveurs les plus performants sont ceux qui raffinent l'algorithme ***MCTS*** avec des méthodes de l'apprentissage machine tels que l'apprentissage par renforcement couplé à réseaux de neurones convolutionnels. Nous n'avons pas cherché à implémenter de tels algorithmes mais nous nous proposons ici d'en faire une courte présentation.

Apprentissage par renforcement

L'apprentissage machine, également connu sous le nom d'apprentissage automatique ou "machine learning", est un domaine de l'intelligence artificielle qui utilise des algorithmes pour permettre à un ordinateur de "s'entraîner" à partir de données et de prendre des décisions en fonction de cette expérience.
Selon les informations disponibles durant la phase d'apprentissage, l'apprentissage est qualifié de différentes manières. On parle généralement d'apprentissage supervisé, d'apprentissage non supervisé et d'apprentissage par renforcement lorsqu'on veut effectuer une subdivision de ce domaine.

L'apprentissage par renforcement (reinforcement learning ou RL) est une méthode d'apprentissage automatique qui consiste à entraîner un agent à effectuer des actions dans un environnement en fonction des récompenses qu'il reçoit. Contrairement à l'apprentissage supervisé, l'agent n'a pas de données étiquetées pour l'aider à apprendre. Au lieu de cela, l'agent explore son environnement en prenant des actions et en recevant des récompenses pour celles-ci. L'objectif de l'agent est de maximiser sa récompense globale sur la durée de l'interaction avec l'environnement.

L'apprentissage par renforcement est utilisé dans de nombreux domaines, notamment dans les jeux, la robotique, la finance, la santé, la gestion des ressources naturelles, et bien d'autres encore. Dans le cadre des jeux, l'apprentissage par renforcement peut être utilisé pour entraîner un agent à jouer à des jeux tels que les échecs, le Go, ou encore Dots and Boxes, comme indiqué plus haut.

Un algorithme de RL dans notre cas aura pour objectif d'attribuer une certaine mesure de performance à chaque coup que l'on peut noter formellement comme

une fonction de s et de a noté $Q(s,a)$. A chaque état s, on cherche à actualiser la valeur de $Q(s,a)$ selon la formule suivante:

$$Q[s,a] := (1-\alpha)Q[s,a] + \alpha\left(r + \gamma \max_{a'} Q[s',a']\right)$$

où α représente la vitesse d'apprentissage du système, autrement dit il encode la corrélation entre l'ancienne valeur de $Q(s,a)$ et la valeur actualisée, et ou γ est le discount factor du système, qui permet de piloter l'impact de la récompense sur l'actualisation de $Q(s,a)$ et r est la récompense reçue par l'agent.

L'apprentissage profond

L'apprentissage profond est un sous-domaine de l'apprentissage machine qui utilise des réseaux de neurones artificiels pour apprendre à partir de données non structurées ou semi-structurées. Cette technologie est capable de traiter de grandes quantités de données et d'extraire des caractéristiques pertinentes, permettant ainsi de résoudre des problèmes complexes tels que la reconnaissance d'images, la compréhension du langage naturel et la prise de décisions en temps réel. L'apprentissage profond est considéré comme une avancée majeure dans le domaine de l'intelligence artificielle et est utilisé dans une variété d'applications.

Les avancées dans le domaine de l'apprentissage profond (ou deep learning) ont également accéléré les progrès en RL avec l'utilisation d'algorithmes d'apprentissage profond dans ce domaine ce qui a aboutit à un nouveau domaine qui est celui de l'apprentissage par renforcement profond (deep reinforcement learning ou DRL). Ici, des réseaux neuronaux sont utilisés pour approximer la politique ou la fonction de score optimale, permettant au RL de s'étendre à des problèmes de prise de décision qui étaient auparavant inextricables en raison de l'étendue des espaces d'état et d'action. C'est également le cas pour AlphaGo Zero et AlphaZero, qui sont tous deux des systèmes DRL. En enrichissant l'approche RL avec des réseaux neuronaux profonds pour l'estimation de la politique et de la valeur en combinaison avec un algorithme de recherche heuristique, AlphaGo Zero et AlphaZero sont capables de surpasser les approches d'IA de jeu avec des heuristiques fabriquées à la main et de battre les meilleurs joueurs du monde aux échecs et au Go.

Cas Alpha Zero

AlphaZero est un système d'apprentissage par renforcement profond développé par la société britannique DeepMind en 2017. L'utilisation de AlphaZero est pertinente pour résoudre le jeu Dots and Boxes car AlphaZero est une méthode

d'apprentissage par renforcement qui permet de développer des stratégies de jeu à partir de zéro, sans nécessiter de connaissances préalables sur le jeu si ce n'est que les règles. En jouant contre lui-même, AlphaZero apprend progressivement les schémas et les tactiques qui mènent à la victoire dans ce jeu complexe. Avec suffisamment de temps et de ressources de calcul, AlphaZero est capable éventuellement développer des stratégies imbattables pour Dots and Boxes.

L'utilisation d'AlphaZero présente également l'avantage de pouvoir être appliquée à des variantes de Dots and Boxes, telles que des grilles non carrées ou des grilles avec des diagonales, car l'algorithme n'a pas besoin de connaître à l'avance les stratégies gagnantes pour chaque variante. Cela permet d'explorer de nouvelles variantes du jeu et d'étudier leur complexité.

En résumé, l'utilisation de AlphaZero est pertinente pour résoudre le jeu Dots and Boxes car cette méthode d'apprentissage par renforcement permet de développer des stratégies de jeu optimales sans nécessiter de connaissances préalables sur le jeu ou des heuristiques prédéfinies. De plus, cette approche peut être étendue à des variantes du jeu, ce qui permet d'explorer la complexité de ce jeu sous différents angles.

V. Interprétation des résultats

Pour évaluer la performance des différents solveurs que nous avons implémentés, nous nous proposons de les faire jouer les uns contre les autres. Pour éviter des temps de calculs trop importants mais pour tout de même obtenir des résultats exploitables nous choisissons d'appliquer le protocole suivant: chaque solveur affronte l'ensemble des autres algorithmes pour un total de cent parties. Les données que nous choisissons d'exploiter sont le pourcentage de victoires de chaque solveur, ainsi que le temps de calcul nécessaire à l'exécution de tous les matchs. Nous étudions également l'influence de différents paramètres sur la performance ainsi que le temps de calcul nécessaire pour chaque solveur comme la profondeur de recherche (ou depth) pour MiniMax, le nombre de simulations effectuées (ou le nombre de rollouts) pour MCTS, ou la taille du plateau de jeu (le nombre de lignes et de colonnes).

Tout d'abord les temps de calculs des algorithmes sont donnés par les graphes suivants :

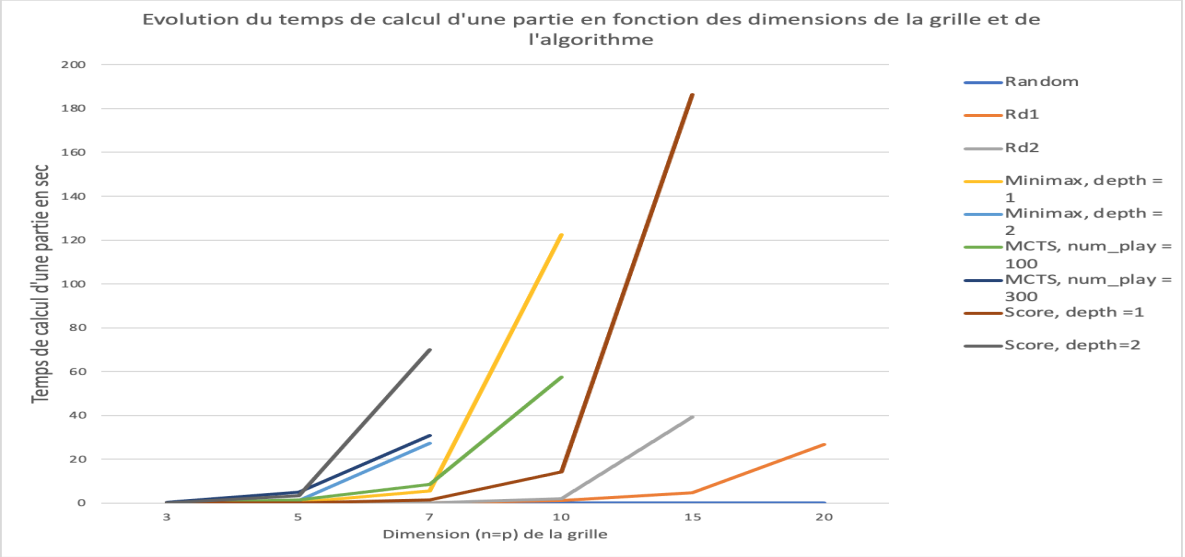

Figure 11: Temps de calcul pour une partie en fonction de l'algorithme et de la taille de grille

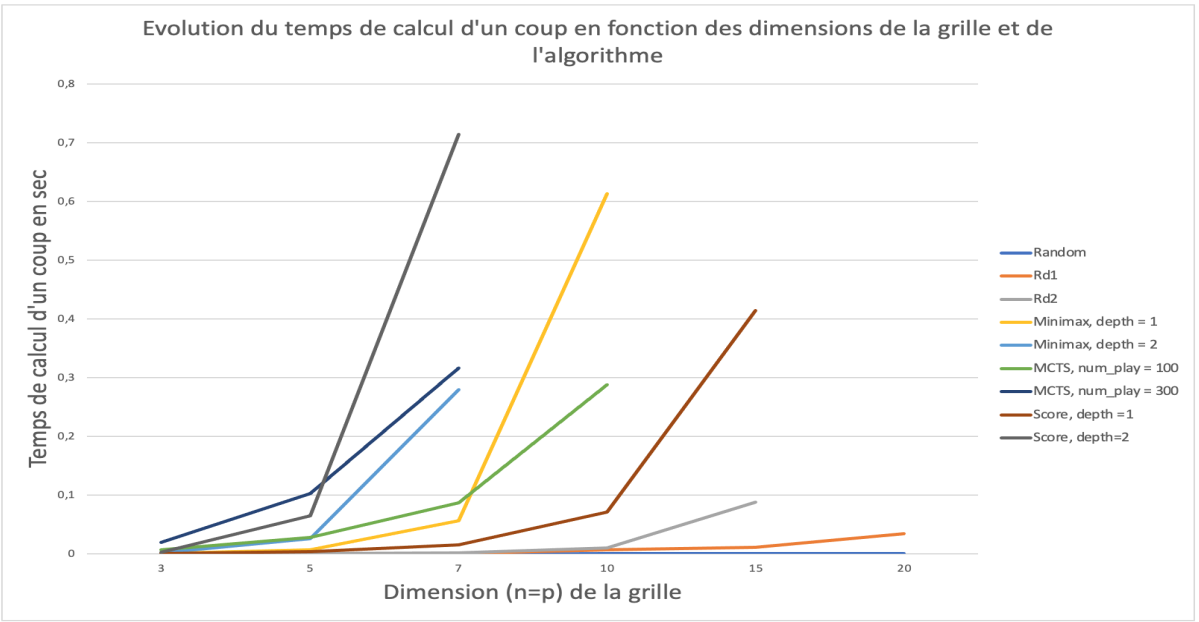

Figure 12: Temps de calcul pour un coup en fonction de l'algorithme et de la taille de grille

Comme l'indiquent les graphes, très peu d'algorithmes terminent rapidement sur grandes grilles (n,p>10) : Random1, Random2 et Score (et Random qui a un intérêt très limité).

Les résultats des duels entre les différents algorithmes sont donnés dans les matrices suivantes. La **valeur indiquée désigne la probabilité que l'algorithme correspondant à la ligne batte l'algorithme correspondant à la colonne**. L'algorithme de score de profondeur 2 n'est pas représenté car il obtient des résultats faibles malgré des temps de calcul très importants dès n,p=4.

Dans le cas n=3 et p=3 nous obtenons les performances suivantes :

	Random	Rd1	Rd2	Minimax, depth = 1	Minimax, depth = 2	MCTS, num_play = 100	MCTS, num_play = 300	Score, a=1	Score, a=100
Random	0,5	0,29	0,28	0,16	0,3	0,25	0,12	0,5	0,5
Rd1	0,71	0,5	0,38	0,4	0,3	0,64	0,55	0,6	0,48
Rd2	0,72	0,62	0,5	0,55	0,45	0,65	0,7	0,7	0,57
Minimax, depth = 1	0,84	0,65	0,57	0,5	0,4	0,65	0,7	0,73	0,66
Minimax, depth = 2	0,7	0,82	0,55	0,6	0,5	0,77	0,87	0,96	0,83
MCTS, num_play = 100	0,75	0,36	0,35	0,35	0,23	0,5	0,5	0,44	0,31
MCTS, num_play = 300	0,88	0,45	0,3	0,3	0,13	0,5	0,5	0,5	0,5
Score, a=1	0,5	0,4	0,3	0,27	0,04	0,56	0,5	0,5	0,5
Score, a=100	0,5	0,62	0,43	0,34	0,17	0,69	0,5	0,5	0,5

Dimensions n = 3, p = 3

Figure 13: Résultats des duels d'algorithmes pour n,p = 3

Les résultats de l'algorithme MCTS sont surprenants et ne sont pas en accord avec nos prévisions. Nous nous attendions à une performance bien supérieure pour un nombre de simulations équivalents. Nous pouvons formuler plusieurs hypothèses pour expliquer ces faibles résultats. La principale étant qu'il existe un nombre de simulations critiques au-delà duquel la performance de l'algorithme explose.
Cependant, le temps de calcul devient très important assez rapidement lorsque l'on augmente le nombre de playout ce qui empêche d'avoir un algorithme MCTS efficace.

Pour n,p=5 et pour les algorithmes retenus (résultats intéressants et temps de calcul convenable) nous avons les résultats suivants :

	Random	Rd1	Rd2	Minimax, depth = 1	Minimax, depth = 2	Score, a=1	Score, a=100
Random	0,5	0	0	0	0	0	0
Rd1	1	0,5	0	0	0	0,75	0,75
Rd2	1	1	0,5	0,37	0,37	0,75	0,62
Minimax, depth = 1	1	1	0,63	0,5	0,5	0,75	0,75
Minimax, depth = 2	1	1	0,63	0,5	0,5	0,62	0,87
Score, a=1	1	0,25	0,25	0,25	0,38	0,5	0,5
Score, a=100	1	0,25	0,38	0,25	0,13	0,5	0,5

Dimensions n = 5, p = 5

Figure 14: Résultats des duels d'algorithmes pour n,p = 5

L'algorithme Minimax (depth = 1 et depth = 2) domine tous les autres algorithmes étudiés. En effet, il y a une croissance stricte de la qualité des algorithmes entre les algorithmes Random 1, Random 2 et Minimax.

Malgré des taux de victoire assez faibles, l'algorithme score parvient tout de même à résister face à tous les algorithmes et atteint notamment un taux de victoire de 38% face à Minimax depth = 2 malgré un temps de calcul 9 fois moins long pour n,p = 5.

Enfin, l'écart de résultats entre les deux algorithmes Minimax est étonnamment assez faible. L'algorithme le plus intéressant de tous semble donc être Minimax depth = 1 (qui est 4 fois plus rapide que depth = 2 pour n = 5).
L'algorithme Minimax de profondeur 1 a en effet de très bons résultats et est relativement rapide puisque chaque coup a un temps de calcul inférieur à 1 seconde pour n,p <= 10.

VI. Études théoriques supplémentaires

a) Partie à 3 joueurs

Nous nous proposons dans cette partie d'étudier dans un cadre formel le jeu dots and boxes extrapolés à trois joueurs (et à un nombre arbitraire N joueurs). A notre connaissance, ces variantes n'ont jamais fait l'objet de publications scientifiques. Notre objectif est d'observer si les stratégies utilisées dans le cadre du jeu à deux joueurs sont applicables, adaptables ou généralisables à un nombre arbitraire de joueurs. En effet, étant donné que le jeu dots and boxes possède la caractéristique singulière d'être impartial, le fait de jouer à plus que deux joueurs est tout à fait possible en gardant les mêmes principes de jeu. Chaque joueur joue chacun son tour en cherchant à maximiser son nombre de boîtes capturées.

L'ajout d'un troisième joueur vient bouleverser complètement les stratégies heuristiques développées par Berlekamp. En effet, pour parvenir à garder le contrôle, il faut être capable de forcer les deux prochains joueurs à jouer.
Alors qu'à deux joueurs, lorsqu'un joueur était en contrôle de la partie, il devait créer des situations de *hard-hearted handout* (cf I.) pour forcer le joueur suivant à jouer. Lorsqu'il y a plusieurs joueurs, c'est le coup du joueur précédent notre tour qui est intéressant, et non celui du joueur suivant.
Ainsi, alors qu'il fallait éviter les situation de *half-hearted handout*, elles se révèlent utiles à 3 joueurs. En effet, pour le joueur A cela permet de forcer le joueur C, dans la mesure où B n'a pas intérêt à jouer.

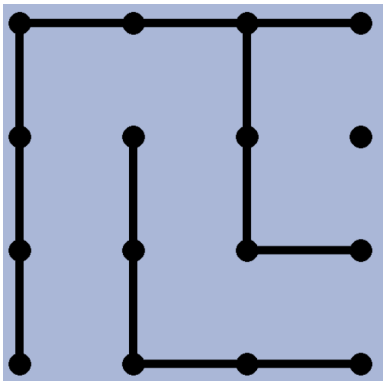

Figure 15: Utilisation du half-hearted handout *pour une partie à 3 joueurs.*

Si c'est au joueur A de jouer, il n'a pas intérêt à créer une situation de *hard-hearted handout* sinon il est sûr de perdre, tandis qu'il a une chance qui dépend du coup de B en créant un *half-hearted handout*.

La grande différence avec le jeu à deux réside dans le *double-dealing*, celui-ci ne permet plus de garder le contrôle. Le joueur A décide simplement s'il le donne à B ou à C.

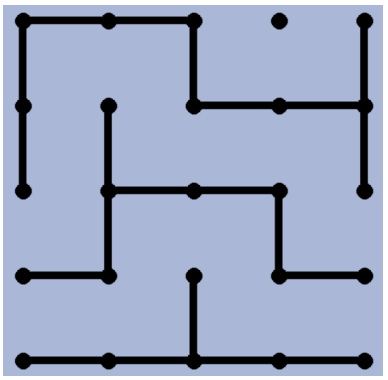

Figure 16: Utilisation du hard-hearted handout *pour une partie à 3 joueurs.*

En revanche, dans la situation ci-dessus, le joueur A a plutôt intérêt à forcer le joueur B à jouer et à déclencher une chaîne. Si l'on considère que le joueur B tente de minimiser le score de C qui joue après, il va déclencher la chaîne la plus courte. C n'ayant pas de moyen de gagner la complète jusqu'au bout pour maximiser son score. A peut donc gagner en complétant la plus longue.

Finalement, en effectuant le même raisonnement que pour la règle de la *long chain* et en supposant que les chaînes se remplissent par ordre croissant de longueur. Le joueur A doit chercher à maximiser la longueur des chaînes qui seront à compléter sur les tours de numéro 1 modulo 3. Pour ce faire il a deux options : compléter sa chaîne en prenant le maximum de carré ou réaliser un

double-dealing pour décaler de un tour le compte. Bien sûr B et C ont le même but pour les tours 2 et 0 modulo 3 avec les mêmes outils.
Ainsi, il ne peut pas y avoir véritablement de stratégie sur le nombre de *long chain* à former car la marche à suivre dépend également de leur longueur.

Quelques résultats expérimentaux dans le cas d'une partie à trois joueurs. Nous testons ici les performances de nos différents agents lorsqu'ils s'affrontent dans une partie à trois joueurs. Nous nous intéressons surtout à l'ordre des joueurs plutôt qu'à la taille de la grille. Les valeurs suivantes sont les proportions de victoires sur 100 affrontements sur des grilles 5x5.

Rd	Rd2	Minimax
0	0,92	0,08
Rd	Minimax	Rd2
0	0,93	0,07

Les résultats ci-contre indiquent que lorsqu'un agent est faible (ici le Random), cela profite au joueur suivant. Même s'il a un comportement plutôt naïf comme Random2. Ainsi bien joué n'est pas nécessairement utile si le joueur précédent n'est pas bon.

Minimax	Rd2	Rd2
0,33	0,23	0,44
Rd2	Minimax	Rd2
0,5	0,33	0,17

Ce résultat illustre bien la façon de procéder de Minimax. Il minimise le score du joueur qui le suit peut-être à son désavantage. Seulement l'implémentation de Minimax est spécifique à une situation à deux joueurs. Les deux résultats montrent que ce n'est pas le fait de jouer le premier qui importe. Cette situation profite au dernier joueur, celui qui joue avant Minimax.

Score	Rd2	Minimax
0	1	0
Score	Minimax	Rd2
0	1	0

Ici, on retrouve la même situation que sur les premiers tests. Celui qui gagne est celui qui joue après l'agent faible.

Rd2	Rd2	Rd2
0,46	0,25	0,29

Ce dernier résultat semble indiquer que pour cet agent, jouer en premier constitue un avantage.

b) Grille à base triangulaire

Nous nous sommes également intéressés à des stratégies humaines sur une variante de grille, celle à base triangulaire. Nous nous intéressons aux grilles de la forme suivante avec m lignes de n points. Ci-dessous nous avons une grille 6×5.

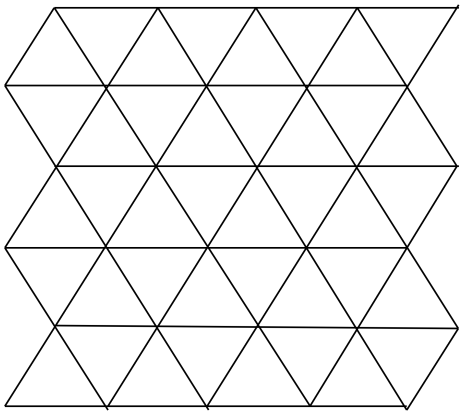

Figure 17: Grille triangulaire 6x5.

Pour une telle grille, nous avons $2(m-1)(n-1)$ triangles. Cela se voit en supprimant toutes les arêtes "diagonales descendantes". On se ramène à une grille de parallélogramme et donc une grille presque carré de taille $m \times n$. De la même façon, en partant d'une grille carrée, on ajoute une diagonale par carré pour obtenir cette grille. Ainsi on a $m(n-1) + n(m-1) + (m-1)(n-1)$. Le terme $(m-1)(n-1)$ en plus des grilles carrés s'annule avec le facteur 2 dans le nombre de triangles donc on obtient la même relation que pour la grille carrée :

Nb de tour = Nb de points + Nb de coups doubles

De même, la notion de chaîne est conservée, ainsi que les situations de *hard/half-hearted handouts* et donc le *double-dealing*. La seule différence est dans les embranchements possibles pour les chaînes. Cela peut changer seulement leur compte mais pas leurs propriétés. La règle de la *long chain* est donc conservée.

VII. Conclusion et prolongements

Ce PSC aura été l'occasion pour nous de découvrir l'étude informatique et mathématique formelle d'un jeu de société, domaine qui ne nous était pas familier. Nous avons pris plaisir à travailler sur un sujet riche et actuel, où de nouvelles publications sont publiées régulièrement. Nous avons ainsi pu nous

immerger pleinement dans ce sujet de recherche, en ayant une vue d'ensemble de sa littérature.
Nous aurions aimé avoir eu le temps d'implémenter l'algorithme AlphaZero pour pouvoir le comparer à nos autres solveurs.
Un axe d'approfondissement serait aussi de créer une policy qui selon l'avancement de la partie utiliserait tel ou tel algorithme. En effet, il apparaît par exemple que les algorithmes coûteux en temps apportent une faible plus value en début de partie et qu'un algorithme comme score est très utile en milieu de partie.
Il est aussi amusant de noter que des stratégies humaines peuvent être le résultat de calculs probabilistes des algorithmes, alors que ces stratégies ont été conçues grâce aux concepts de contrôle de la partie, comme par exemple le *double-dealing* qu'emploie Minimax.
Les résultats montrent que ce jeu reste très complexe puisque les solveurs ne sont clairement pas infaillibles. D'après la littérature le plus performant est celui produit par de l'apprentissage par renforcement mais il est bien moins compréhensible.

VIII. Répartition du travail

Les tâches ont été réparties de la manière suivante :

- Implémentation du jeu et de l'interface graphique : Louis Collomb, Dimitri de Saint Guilhem.
- Rendre possible le jeu à plus de 2 joueurs : Louis Collomb
- Réaliser des grilles non-rectangulaires : Dimitri de Saint Guilhem.
- Développement des différentes policy : Titouan Borderies, Wilfried Diby
- Approfondissement de la documentation et recherche d'une fonction de score à partir de stratégies "humaines": Axel Benyamine.
- Création de la policy associée à la fonction de score : Axel Benyamine
- Développement d'une fonction de score à base d'un réseau de neurones : Titouan Borderies, Wilfried Diby.
- Étude théorique pour des grilles triangulaires ou des potentielles stratégies à 3 joueurs : Dimitri de Saint Guilhem.

Bibliographie :

- [1] Elwyn Berlekamp, *The dots and Boxes game*, A K Peters.
- [2] Jonathan C. T. Kuo, Artificial Intelligence at Play - Connect Four (Mini-max algorithm explained), Medium website.
- [3] Alexandre Sierra Ballarín, Combinatorial Game Theory: The Dots-and-Boxes Game (Master of Science Thesis).

- [4] Dürhsen M., Fabris I., Feyzi F., Gauthy L., Simon E., Giepmans C., *Artificial, Intelligent Agents for Dots and Boxes*, Maastricht University, Department of Datascience and Knowledge Engineering.
- [5] Joseph K. Barker, Richard E Korf, *Solving Dots-And-Boxes,* Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence.
- [6] Abel Romer, *Winning dots-and-Boxes in two and three dimensions,* Submitted in partial fulfillment of the requirements for the Degree of Bachelor of Arts and Sciences Quest University Canada.
- [7] Matthew Rea, *Dots and Boxes*, Cardiff University, Department of Computer Science and Informatics.