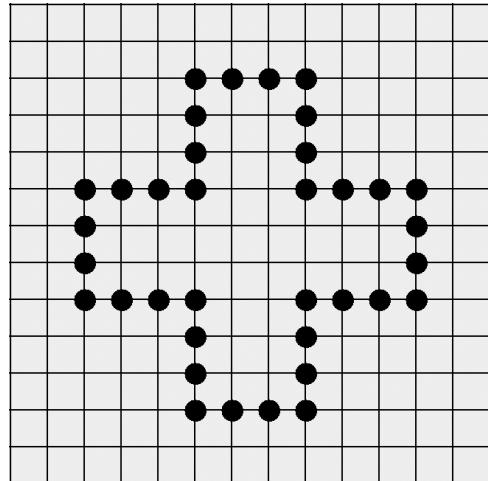


Projet de Recherche Monte Carlo et Jeu

Monsieur CAZENAVE Tristan

Jeu du Morpion Solitaire

Projet réalisé par NDIAYE Maïrame et VED Olesia



Master 2 Intelligence Artificielle, Systèmes et Données

23 juillet 2023

Table des matières

1	Introduction	1
1.1	Présentation du jeu du Morpion Solitaire	1
1.2	Historique de records et informatique	2
2	Développement du jeu	4
2.1	Architecture du projet	4
2.2	Prise en main utilisateur et Interface graphique	6
3	Implémentation des méthodes Monte Carlo	7
3.1	Nested Monte Carlo Search	7
3.2	Nested Rollout Policy Adaptation	8
3.3	Stratégies d'optimisation	9
4	Expérimentations, comparaisons et résultats	10
4.1	Comparaison en terme de qualité des solutions trouvées	10
4.2	Comparaison en terme de complexité temporelle	13
5	Améliorations possibles	14
6	Conclusion	14

1 Introduction

Ce rapport présente le travail que nous avons réalisé dans le cadre du projet de Recherche Monte Carlo et Jeu. L'objectif de ce projet était d'appliquer différentes méthodes de recherche Monte Carlo, vues en cours, à un jeu de notre choix. Le projet s'est divisé en deux parties distinctes : le développement du jeu et l'implémentation des méthodes Monte Carlo, ainsi que leur application au jeu. Notre choix s'est porté sur le jeu du Morpion Solitaire.

1.1 Présentation du jeu du Morpion Solitaire

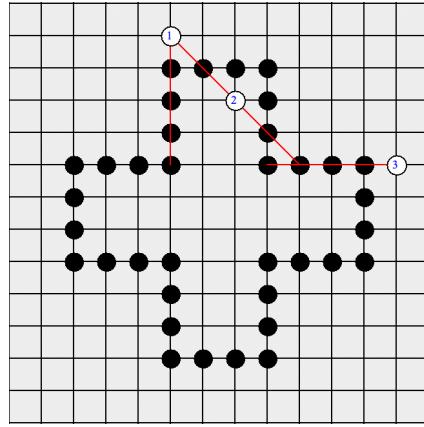


FIGURE 1 – Morpion Solitaire

En avril 1974, Pierre Berloquin présenta le jeu du Morpion Solitaire dans le magazine *Science & Vie* en ces termes : « *Voici un passe-temps qui risque de coûter plus cher aux bureaux et aux administrations que la grippe, le téléphone et les cocottes réunis* ».

Le **Morpion Solitaire** est un casse-tête qui se joue seul. Les règles sont simples : au départ, une grille est donnée avec un ensemble de points déjà tracés, formant généralement une croix grecque dont chaque côté comporte 4 points.

À chaque tour, le joueur doit placer un nouveau point sur la grille de manière à former un alignement de cinq points adjacents, que ce soit horizontalement, verticalement ou en diagonale. Une fois un alignement formé, une ligne est tracée pour le représenter et le score est incrémenté. L'objectif est de placer le plus de points possibles avant d'atteindre un état où il n'est plus possible d'ajouter de nouveaux points.

Deux modes de jeu existent : *5D* ou *5T*. Dans la version *5T*, deux lignes dans la même direction peuvent se toucher uniquement à leurs extrémités. Dans la version *5D*, deux lignes dans la même direction ne peuvent se toucher en aucun point.

La figure 1 illustre l'état du jeu après trois coups.

1.2 Historique de records et informatique

Version 5T

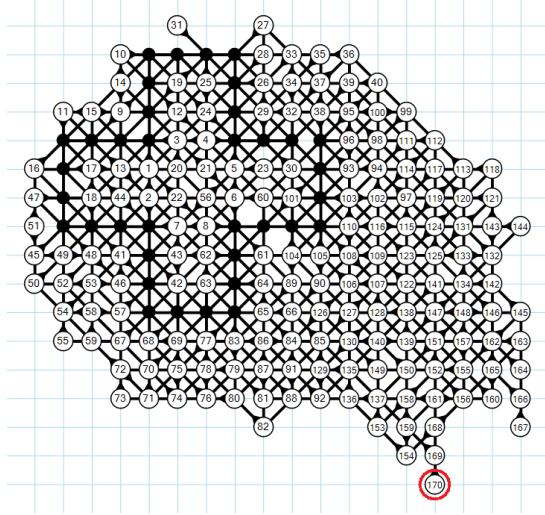


FIGURE 2 – Grille Record Bruneau : 170 coups

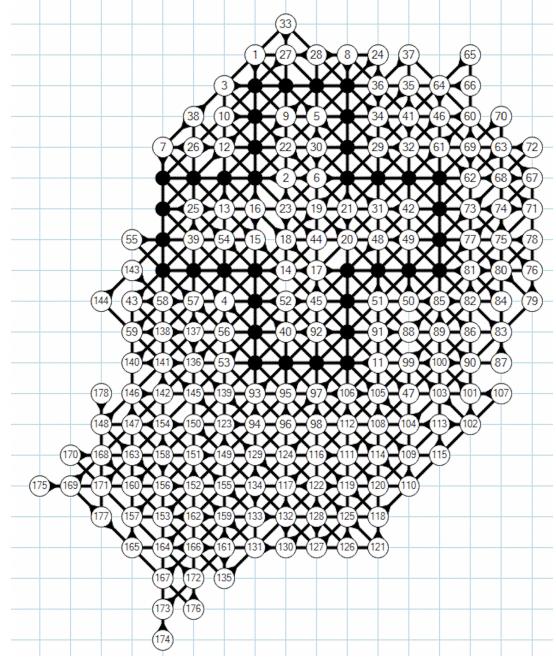


FIGURE 3 – Grille Record Rosin : 178 coups

Le record *manuel* du jeu a été établi par Charles Bruneau en 1976, avec une grille de 170 coups. Ce record a tenu pendant plus de 34 ans avant d'être battu en 2010 par Christopher Rosin, qui a réalisé une grille de 172 coups à l'aide d'un *programme informatique* basé sur les techniques de recherche Monte Carlo. Pour atteindre ce record, Christopher Rosin a combiné l'apprentissage de la politique avec le machine learning et les recherches imbriquées, telles que présentées dans les travaux de Tristan Cazenave en 2009 [2]. En avril 2011, Christopher Rosin a battu son propre record en réalisant une grille de 177 coups, qu'il a à nouveau surpassée trois mois plus tard avec une grille de 178 coups, en utilisant l'algorithme de recherche arborescente NRPA (Nested Rollout Policy Adaptation) [3, 5].

Version 5D

En 2008, Tristan Cazenave a établi un record de 80 coups en utilisant la technique Monte Carlo, notamment l'algorithme NMCS (Nested Monte Carlo Search), qu'il a présenté dans son papier de 2009 [2]. Ce record a été battu en 2010 par Christopher Rosin qui atteint une grille de 82 coups grâce à la combinaison du NMCS avec l'apprentissage d'une politique avec le machine learning.

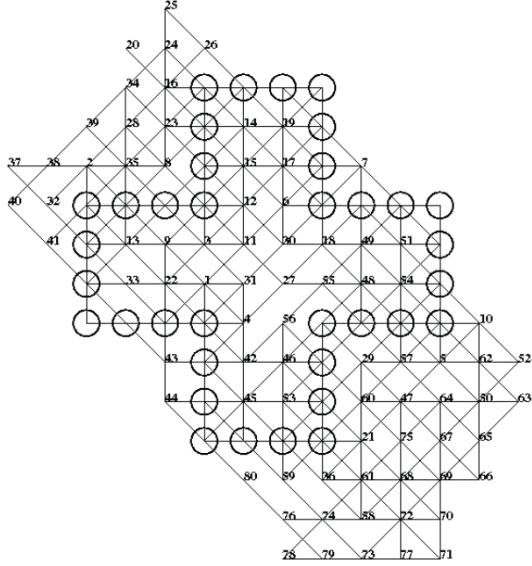


FIGURE 4 – Grille Record Cazenave : 80 coups

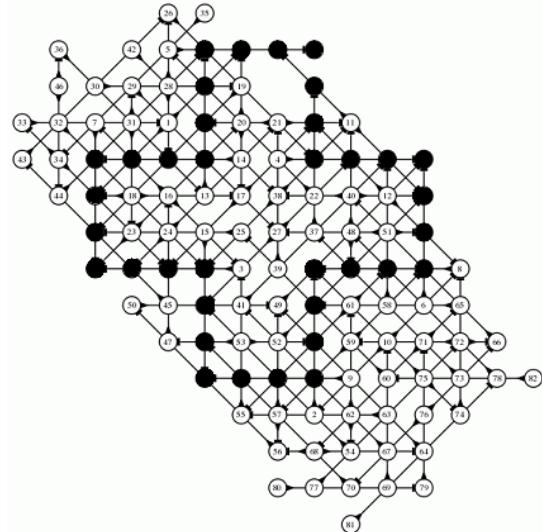


FIGURE 5 – Grille Record Rosin : 82 coups

Ainsi, les techniques de recherche arborescente ont grandement contribué à la recherche de séquences longues pour le Morpion Solitaire, qui a été démontré comme un problème **NP-Difficile** en 2006 [4], se situant à des niveaux de complexité similaires à ceux du jeu de Go.

Pour la réalisation de ce projet, nous avons opté pour l'utilisation du langage Java, qui offre de meilleures performances que Python et que nous maîtrisons davantage que C++. Notre implémentation du jeu se base sur la version 5T du Morpion Solitaire.

Dans la suite de ce rapport, nous commencerons par présenter l'architecture du projet, en détaillant les différentes classes et méthodes que nous avons développées pour le jeu. Ensuite, nous aborderons l'implémentation des méthodes Monte Carlo, ainsi que les stratégies que nous avons mises en place pour améliorer les performances.

Par la suite, nous présenterons les résultats de nos expérimentations et comparaisons réalisées. Nous analyserons les performances de chaque méthode et discuterons des avantages et des limites.

Enfin, nous conclurons en proposant quelques pistes d'amélioration possibles pour notre projet, en soulignant les points qui pourraient être approfondis ou étendus pour obtenir de meilleurs résultats.

2 Développement du jeu

Dans cette section, nous allons présenter les choix que nous avons faits pour le développement du jeu, en mettant l'accent sur le découpage en classes et en méthodes. Ensuite, nous aborderons l'interface graphique du jeu à travers un guide de prise en main utilisateur.

2.1 Architecture du projet

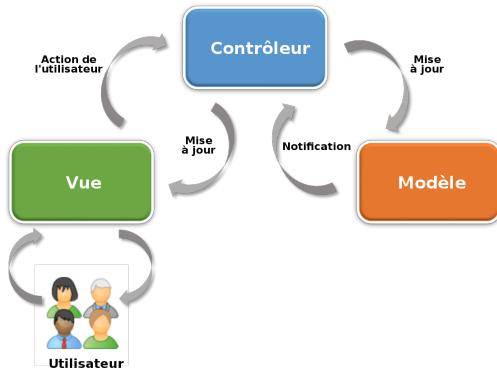


FIGURE 6 – Architecture MVC

Étant donné la nature de notre projet, qui consiste à implémenter un jeu avec une interface graphique, nous avons pris la décision d'utiliser l'architecture **Modèle-Vue-Contrôleur (MVC)** pour simplifier le développement et faciliter la gestion du code. Cette architecture permet de séparer clairement la logique de présentation, la gestion des données et les actions de contrôle en trois composants distincts :

- *Le Modèle* : Il représente la structure des données de notre jeu. Cela inclut la logique métier, la manipulation des données et les opérations effectuées sur celles-ci.
- *La Vue* : C'est la partie responsable de l'affichage des données à travers l'interface utilisateur. Elle est chargée de présenter les informations au joueur de manière appropriée et de répondre à ses interactions.
- *Le Contrôleur* : Il gère les interactions entre l'utilisateur et l'application. Cela inclut la réception des entrées de l'utilisateur, la coordination des actions nécessaires et la communication avec le modèle et la vue.

Pour maintenir une structure organisée, notre code source est divisé en trois packages correspondant à ces trois composantes de l'architecture MVC. Chaque package contient les classes et les fonctionnalités liées à son rôle spécifique dans le modèle de conception.

Package model

Le package contient principalement la classe **Point**. Comme son nom l'indique, cette classe permet de modéliser un point dans le jeu. Un point est caractérisé par trois attributs distincts :

- « id » : qui permet d'identifier à quel moment le point a été joué. Si l'id est 0 alors c'est qu'il s'agit d'un point de la grille initiale.
- Coordonnées « x » et « y » : qui déterminent la position du point sur la grille.
- « tags » : une liste, initialement vide, qui contient les identifiants des points qui ont été placés dans la grille et qui ont permis à ce point de faire partie d'un alignement de cinq points.

Par exemple, considérons la grille de la figure 7. Le point 1 numéroté en vert porte pour tags [1,2] car il s'agit des points placés par l'utilisateur qui lui ont permis de former un alignement vertical et horizontal. De la même façon, le point 4 numéroté en vert porte le tag [1] car c'est le point qui lui permet d'avoir un alignement vertical. Le point 5 numéroté en vert porte le tag [2] qui est le point qui lui a permis d'avoir un alignement horizontal, etc...

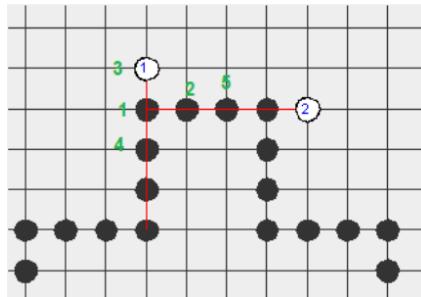


FIGURE 7 – Exemple explicatif tags

Avec cette logique de tag, si cinq points ont un tag en commun, c'est qu'ils sont reliés dans le jeu par une ligne.

Package view

Ce package est composé de cinq classes permettant de gérer les différentes fenêtres de l'interface graphique.

- **Home** : Affiche la page d'accueil lorsque l'utilisateur lance le jeu. Elle présente les options à l'utilisateur pour démarrer une nouvelle partie ou lancer un des solveurs d'intelligence artificielle pour jouer au jeu.
- **GameHuman** : Permet à l'utilisateur de saisir son nom pour commencer à jouer.
- **ShowPlateau** : Responsable de l'affichage du plateau de jeu en fonction de son état.
- **GameInterface** : Gère l'interface de jeu principale pour l'utilisateur. Elle met à jour l'interface graphique pour refléter les mouvements du joueur et l'évolution du jeu.
- **ClassementView** : Gère l'interface graphique qui affiche le classement des scores des différents utilisateurs. Elle présente les meilleurs scores dans un tableau et permet à l'utilisateur de consulter la grille obtenue par le meilleur joueur.

Package controller

Ce package se compose de trois classes principales :

- **Game** : Cette classe regroupe des méthodes essentielles pour la gestion du jeu. La méthode la plus importante de cette classe est *pointPlayable()*, qui prend en entrée la liste des points déjà présents sur la grille ainsi qu'un nouveau point, et renvoie un alignement de cinq points qui permettrait de placer le nouveau point. Si aucun alignement valide n'est possible, la méthode renvoie null.

La stratégie pour déterminer si un point est jouable est la suivante :

1. Pour chaque direction (horizontale, verticale, diagonale et anti-diagonale), on récupère dans une liste les 4 points existants à gauche du point que l'on souhaite placer, ainsi que les 4 points existants à droite (s'il n'y en a que deux, on prend seulement ces deux). Si l'une des listes des quatre directions contient plus de 5 points (en incluant le nouveau point), cela suggère qu'un alignement de 5 points pourrait être possible.
 2. Ensuite, il faut vérifier si cet alignement est valide selon les règles du morpion solitaire 5T. Pour cela, on utilise les "tags" des points. Rappel : Deux points ont le même tag s'ils sont déjà reliés par une ligne dans le jeu. Ainsi, un nouvel alignement est valide selon les règles 5T uniquement si les cinq points de l'alignement n'ont pas de tag en commun. Cela signifie que la nouvelle ligne touche au plus l'extrémité d'une ligne déjà tracée dans la même direction. Cette règle est vérifiée dans la fonction.
 3. Si l'alignement est valide, les cinq premiers points de cet alignement sont renvoyés.
- **PlayablePoint** : Cette classe contient une méthode statique qui permet de déterminer, en fonction de l'état actuel du plateau, tous les points jouables. Cette méthode analyse les points déjà placés sur la grille et utilise la méthode *pointPlayable()* de la classe "Game" pour déterminer si un point peut être joué ou non.
 - **IASolver** : Cette classe contient les implémentations des méthodes *NMCS (Nested Monte Carlo Search)* et *NRPA (Nested Rollout Policy Adaptation)* ainsi que de certaines heuristiques pour jouer au morpion solitaire. Les détails de cette implémentation seront présentés ultérieurement dans la suite du rapport.

2.2 Prise en main utilisateur et Interface graphique

Pour plus de clarté et de facilité de référence, cette partie a été rédigée dans le fichier **GuideUtilisateur.pdf**. Dans cette partie, nous expliquons comment lancer le jeu à partir de la ligne de commande et fournissons un guide à l'utilisateur de l'application. Nous détaillons chacune des fonctionnalités implémentées et présentons les différents écrans que l'utilisateur peut rencontrer lors de son expérience de jeu.

3 Implémentation des méthodes Monte Carlo

Les méthodes Monte Carlo sont particulièrement adaptées pour résoudre des problèmes complexes comme le Morpion Solitaire. Elles permettent de générer des solutions en effectuant des simulations aléatoires pour évaluer les coups possibles et choisir les plus prometteurs. Pour le jeu de Morpion Solitaire, nous avons décidé d'utiliser les méthodes *NMCS* (*Nested Monte Carlo Search*) et *NRPA* (*Nested Rollout Policy Adaptation*). Ces deux méthodes sont considérées comme étant à la pointe de la recherche pour la recherche de solutions à ce jeu, avec le NRPA étant une approche plus récente et avancée par rapport à l'état de l'art. Vous pouvez retrouver notre implémentation de ces deux algorithmes dans la classe *IASolver.java* du package *controller*.

3.1 Nested Monte Carlo Search

Le *NMCS* (*Nested Monte Carlo Search*) est une méthode de recherche basée sur les techniques de Monte Carlo. L'algorithme prend en paramètre un état donné du jeu et un niveau de profondeur de recherche. Au niveau de base, la procédure utilise une recherche aléatoire pour simuler des séquences de coups possibles. Aux niveaux supérieurs, l'algorithme effectue des appels récursifs vers le niveau inférieur pour trouver le coup à jouer. Le coup sélectionné est celui qui maximise le score obtenu au niveau inférieur. Cette approche permet d'optimiser les mouvements à toutes les étapes du jeu, ce qui conduit à une solution finale optimisée. En figure 8 vous retrouvez le pseudo code de l'algorithme.

Algorithme 11 : NMCS

Input : *level*, starting state *s_{start}*
Output : best sequence of actions *seq_{best}*, best solution *x_{best}*

```
1 s  $\leftarrow$  sstart
2 if level = 0 then
3   return RandomSimulation(s)
4 i  $\leftarrow$  1
5 repeat
6   foreach a  $\in$  Actions(s) do
7     s  $\leftarrow$  PerformAction(a, s)
8     (seq, x)  $\leftarrow$  NMCS(level - 1, s)
9     s  $\leftarrow$  UnperformAction(a, s)
10    if F(x) is better than F(xbest) then
11      xbest  $\leftarrow$  x
12      seqbest  $\leftarrow$  seq
13    abest  $\leftarrow$  seqbest[i++]
14    s  $\leftarrow$  PerformAction(abest, s)
15 until IsTerminal(s);
16 return (seqbest, xbest)
```

FIGURE 8 – Pseudo Code NMCS

Le choix du niveau de profondeur dans l'algorithme NMCS est crucial. Il a un impact significatif sur les performances de l'algorithme et sur la complexité temporelle. Un niveau plus élevé permet une recherche plus approfondie et peut conduire à de meilleures solutions, mais cela peut également faire exploser le temps de calcul nécessaire. Le choix du niveau doit donc être équilibré en fonction des contraintes de performance et des exigences spécifiques du jeu. Dans la partie d'expérimentation, nous discuterons plus en détail des performances de l'algorithme NMCS à différents niveaux de profondeur et de son influence sur la complexité temporelle et la qualité des solutions obtenues.

3.2 Nested Rollout Policy Adaptation

Le *NRPA* (*Nested Rollout Policy Adaptation*) est une méthode avancée de recherche basée sur les techniques de Monte Carlo et d'apprentissage par renforcement, qui s'est avérée efficace pour résoudre de nombreux problèmes d'optimisation. Cet algorithme apprend une politique de jeu en effectuant des simulations et en ajustant les poids associés à chaque action (voir figure 9). Pendant la phase de playout (voir figure 10), les actions sont sélectionnées avec une probabilité proportionnelle à l'exponentielle du poids qui leur est associé. L'algorithme itère et utilise une approche récursive pour trouver la meilleure séquence de coups à chaque niveau de profondeur. Ensuite, lors de la phase d'adaptation (voir figure 11), la politique est mise à jour en augmentant les poids des meilleures actions et en réduisant les poids des autres actions en fonction de leurs probabilités d'être jouées. Ainsi, le NRPA équilibre l'exploitation en adaptant les probabilités de jouer les coups en fonction de la meilleure séquence trouvée, tout en assurant une exploration grâce à l'échantillonnage de Gibbs au niveau le plus bas.

Algorithm 3 The NRPA algorithm.

```

1: NRPA (level, policy)
2:   if level == 0 then
3:     return playout (root, policy)
4:   else
5:     bestScore  $\leftarrow -\infty$ 
6:     for N iterations do
7:       (result,new)  $\leftarrow$  NRPA(level - 1, policy)
8:       if result  $\geq$  bestScore then
9:         bestScore  $\leftarrow$  result
10:        seq  $\leftarrow$  new
11:      end if
12:      policy  $\leftarrow$  Adapt (policy, seq)
13:    end for
14:    return (bestScore, seq)
15:  end if
```

FIGURE 9 – Pseudo Code NRPA

Algorithm 1 The playout algorithm

```

1: playout (state, policy)
2:   sequence  $\leftarrow []$ 
3:   while true do
4:     if state is terminal then
5:       return (score (state), sequence)
6:     end if
7:     z  $\leftarrow 0.0$ 
8:     for m in possible moves for state do
9:       z  $\leftarrow z + \exp(\text{policy}[\text{code}(m)])$ 
10:    end for
11:    choose a move with probability  $\frac{\exp(\text{policy}[\text{code}(move)])}{z}$ 
12:    state  $\leftarrow$  play (state, move)
13:    sequence  $\leftarrow$  sequence + move
14:  end while
```

FIGURE 10 – Pseudo Code Playout

Algorithm 2 The Adapt algorithm

```

1: Adapt (policy, sequence)
2:   polp  $\leftarrow$  policy
3:   state  $\leftarrow$  root
4:   for move in sequence do
5:     polp [code(move)]  $\leftarrow$  polp [code(move)] +  $\alpha$ 
6:     z  $\leftarrow 0.0$ 
7:     for m in possible moves for state do
8:       z  $\leftarrow z + \exp(\text{policy}[\text{code}(m)])$ 
9:     end for
10:    for m in possible moves for state do
11:      polp [code(m)]  $\leftarrow$  polp [code(m)] -  $\alpha * \frac{\exp(\text{policy}[\text{code}(m)])}{z}$ 
12:    end for
13:    state  $\leftarrow$  play (state, move)
14:  end for
15:  policy  $\leftarrow$  polp
```

FIGURE 11 – Pseudo Code Adapt

Le NRPA est une méthode plus avancée que le NMCS. Plusieurs paramètres peuvent influencer les performances de l'algorithme, tels que le paramètre alpha, le choix de la

politique initiale, le niveau de profondeur de la recherche, le nombre d’itérations, etc. Dans la partie expérimentation, nous analysons l’impact de ces paramètres sur la qualité des solutions trouvées et sur la complexité temporelle.

3.3 Stratégies d’optimisation

Bien que les méthodes Monte Carlo soient très efficaces pour rechercher une solution optimale au jeu du Morpion Solitaire, elles peuvent être coûteuses en termes de complexité temporelle. Dans cette partie, nous présentons les stratégies que nous avons mises en place pour améliorer le temps d’exécution des algorithmes aux niveaux supérieurs.

Parallélisation : Pour exploiter efficacement les ressources informatiques disponibles, nous avons mis en œuvre des mécanismes de parallélisation en utilisant le multi-threading sur certaines parties du code gourmandes en calcul, telles que le calcul de la liste des points jouables. Ainsi, cela nous a permis d’exécuter plusieurs simulations en parallèle sur différents threads, ce qui a accéléré le processus de recherche des algorithmes.

Optimisation des structures de données : Nous avons optimisé les structures de données utilisées dans les algorithmes pour réduire le temps d’accès et de recherche. Par exemple, nous avons utilisé des structures de données appropriées pour représenter la grille de jeu sous forme de liste de points, les points déjà placés avec un identifiant et les alignements à travers l’utilisation de tags. Cette approche nous permet d’effectuer des opérations de recherche et de manipulation plus efficaces, ce qui réduit la complexité temporelle globale de nos algorithmes.

Utilisation d’heuristique : Pour améliorer la complexité temporelle de nos algorithmes, nous avons exploré l’utilisation d’heuristiques. L’idée était de guider l’algorithme vers les coups les plus prometteurs afin de réduire le nombre d’appels récursifs nécessaires.

Dans des méthodes comme le NMCS, un appel récursif est réalisé pour chaque coup possible à un état donné du jeu. En utilisant une heuristique, nous pouvons prioriser les coups les plus susceptibles de conduire à une solution optimale. Cela permet de réduire le nombre de coups nécessitant des appels récursifs, ce qui se traduit par une amélioration de la complexité temporelle.

La classe *IASolver* du package *controller* contient la méthode *NMCSHeuristique()*, qui est une version du NMCS où nous utilisons une heuristique pour améliorer la complexité temporelle sans trop dégrader les performances de l’algorithme de base. Nous donnons en paramètre de cette méthode un entier correspondant au nombre de coups que l’on souhaite développer dans la boucle principale. Les coups sélectionnés ne sont pas choisis de manière aléatoire ; il s’agit des coups les plus prometteurs. Dans cette heuristique, nous considérons comme coup prometteur celui qui a un potentiel de former beaucoup d’alignements.

Par exemple, si nous mettons en paramètre de la fonction un nombre de coups à développer de 5, alors le NMCS, à chaque appel récursif, explorera les 5 coups qui sont les plus

susceptibles de créer des alignements. Cette heuristique nous permet ainsi de contrôler le nombre de coups à développer, ce qui permet d'améliorer la complexité temporelle. En ajustant le nombre de coups explorés en fonction du niveau de profondeur, nous pouvons adapter l'algorithme à différents scénarios. Par exemple, au niveau 1, nous pourrions explorer un plus grand nombre de coups, tandis qu'au niveau 2, nous pourrions nous restreindre sur cette quantité pour réduire la complexité.

De plus, le fait que les coups les plus prometteurs soient explorés permet de conserver des performances proches de celles de la version de base de l'algorithme. Ainsi, en utilisant cette heuristique, nous avons pu optimiser nos algorithmes sans compromettre la qualité des solutions trouvées.

4 Expérimentations, comparaisons et résultats

Dans cette partie, nous présentons les résultats des expérimentations que nous avons réalisées. Ces tests approfondis nous ont permis d'analyser les propriétés des différents algorithmes et de les comparer en termes de qualité des solutions trouvées ainsi que de complexité temporelle.

Nous avons effectué des essais sur différentes configurations, en ajustant notamment les paramètres tels que le niveau de profondeur des algorithmes, le nombre d'itérations, et l'utilisation éventuelle de l'heuristique que nous avons développée pour le NMCS. Pour chaque test, les algorithmes sont lancés 50 fois et le résultat de chaque lancement a été stockée pour voir clairement les tendances.

4.1 Comparaison en terme de qualité des solutions trouvées

Influence du niveau de profondeur

Comme prévu, nous avons observé que plus le niveau de profondeur est élevé, plus les scores obtenus sont généralement meilleurs. Ces observations ont été confirmées tant pour le NMCS que pour le NRPA.

Pour illustrer ces résultats, nous avons tracé des graphiques présentant les scores obtenus par les algorithmes lors des 50 parties pour différents niveaux de profondeur (voir figure 12 et 13). Pour une meilleure visualisation, nous avons trié les listes de scores dans l'ordre croissant afin de mettre en évidence les fourchettes de scores et les tendances.

L'une des conclusions que nous pouvons tirer de ces graphiques est que plus la profondeur est grande, moins les scores sont variables. Par exemple, pour le NMCS de niveau 0, nous observons clairement le caractère complètement aléatoire de l'algorithme, avec des variations importantes allant de 20 à 80. En revanche, le NMCS de niveau 1 a montré des scores beaucoup plus stables, illustrés par une courbe quasiment horizontale avec peu de fluctuations.

Nous avons également remarqué le même phénomène du côté du NRPA, où une augmentation du niveau de profondeur a conduit à des scores plus stables et plus élevés.

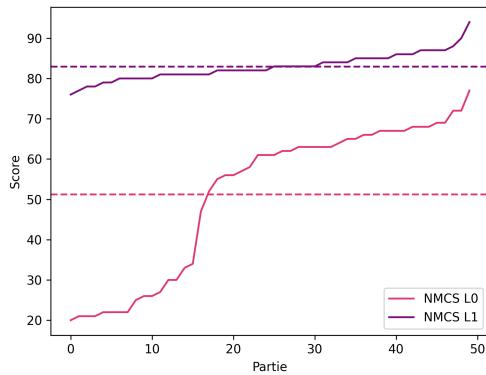


FIGURE 12 – NMCS

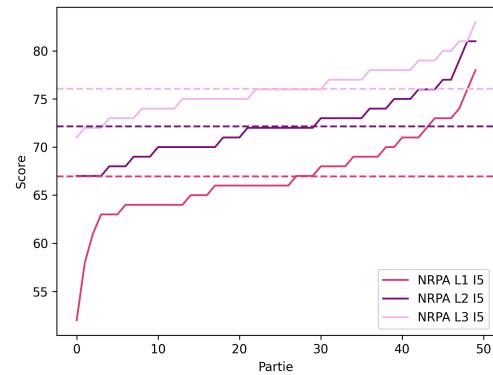


FIGURE 13 – NRPA

Influence du nombre d’itérations sur l’algorithme NRPA

Comme nous l’avions observé avec la profondeur, nous constatons que plus le nombre d’itérations est élevé, plus les scores obtenus sont généralement élevés.

Nous avons illustré ces résultats à l’aide de graphiques, notamment la figure 14 pour un niveau de profondeur de 1 et la figure 15 pour un niveau de profondeur de 2. Ces graphiques mettent en évidence l’augmentation des scores moyens en fonction du nombre d’itérations.

De plus, en observant la figure 14, qui représente les scores moyens en fonction du nombre d’itérations pour le niveau 1, nous pouvons noter un autre phénomène intéressant. En effet, plus le nombre d’itérations augmente, moins l’écart entre les scores moyens est important. Par exemple, l’écart entre le score moyen pour 5 itérations et celui pour 15 itérations est clairement plus grand que celui entre le score moyen pour 100 itérations et celui pour 200 itérations.

Cela nous indique que l’augmentation du nombre d’itérations améliore les performances de l’algorithme, mais qu’à partir d’un certain seuil, cela ne fait plus de différence significative. À ce stade, il serait peut-être plus bénéfique d’envisager d’augmenter la profondeur de l’algorithme pour obtenir de meilleures solutions.

Remarque : Bien que les tests n’aient pas été spécifiquement conçus pour comparer les performances du NMCS et du NRPA, il est intéressant de noter les différences de résultats entre les deux algorithmes. Comme le montrent les figures précédentes, un NMCS de niveau 1 semble donner des solutions plus qualitatives qu’un NRPA de niveau 1 ou de niveau 2. Toutefois, il est important de souligner que ces différences de performances peuvent être dues à la sensibilité des algorithmes aux paramètres choisis, tels que la profondeur et le nombre d’itérations. Le NRPA pourrait potentiellement obtenir des scores comparables à ceux du NMCS en augmentant le niveau de profondeur.

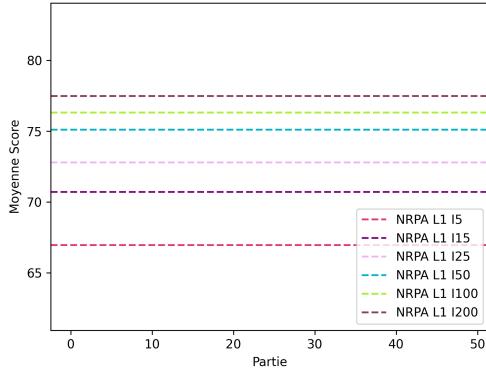


FIGURE 14 – NRPA niveau 1

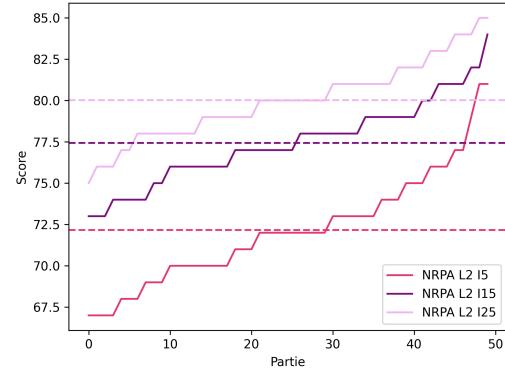


FIGURE 15 – NRPA niveau 2

NMCS de base VS NMCS avec heuristique

Nous avons réalisé un test pour comparer la qualité des solutions obtenues par le NMCS avec ou sans heuristique. Dans la figure 16, nous avons tracé les scores moyens obtenus par les deux algorithmes au niveau de profondeur 1 sur 50 parties.

Comme nous l'attendions, le NMCS de base a obtenu un score moyen plus élevé que le NMCS avec heuristique. Cependant, le NMCS avec heuristique, qui explore 10 coups prometteurs à chaque appel, a un score moyen très proche de celui du NMCS de base, tout en ayant un temps d'exécution réduit.

Une observation intéressante est que plus le nombre de coups explorés est élevé, meilleure est la qualité des solutions. Cependant, nous avons également remarqué que plus le nombre de coups explorés est élevé, moins l'écart entre les scores moyens est important. Cela suggère qu'il existe un seuil au-delà duquel augmenter le nombre de coups explorés n'a plus un impact significatif sur la qualité des solutions.

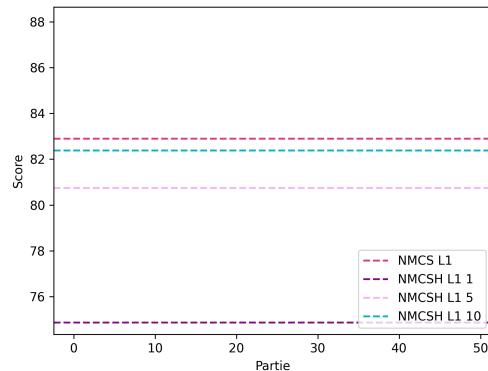


FIGURE 16 – NMCS niveau 1 VS NMCS-H (avec heuristique) niveau 1

Ces observations nous permettent de conclure que l'utilisation de l'heuristique n'a pas considérablement dégradé les performances du NMCS, tout en offrant un gain significatif en termes de complexité temporelle. L'heuristique choisie semble être un compromis efficace pour obtenir des solutions de qualité tout en optimisant le temps d'exécution de l'algorithme.

4.2 Comparaison en terme de complexité temporelle

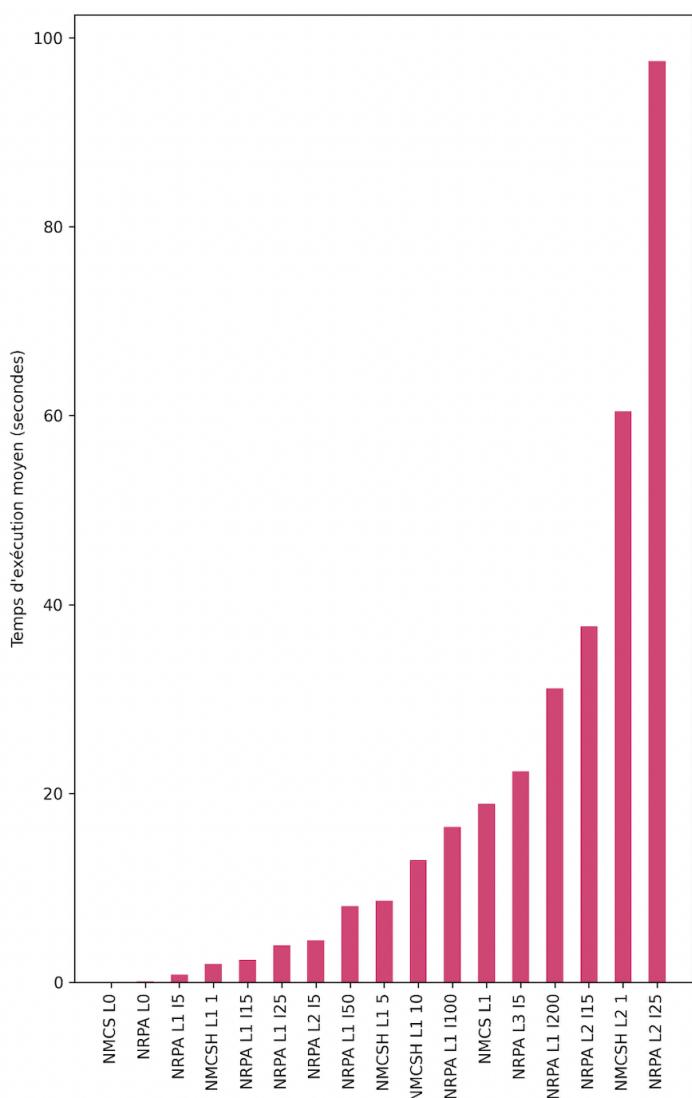


FIGURE 17 – Temps d'exécution des algorithmes

En comparant le NMCS de base au NMCS avec heuristique pour un même niveau, nous observons que le NMCS avec heuristique reste plus rapide en termes d'exécution, tout en préservant une qualité de solutions raisonnable.

Lorsque nous examinons les performances du NRPA pour un même niveau avec un nombre variable d'itérations, nous remarquons que la complexité temporelle augmente de

Les résultats des expérimentations ont mis en évidence un aspect important à prendre en compte : le temps d'exécution des algorithmes. En effet, la qualité des solutions n'est pas le seul critère à considérer ; il est également crucial de mesurer le coût temporel associé à chaque algorithme.

Dans la figure 17, nous avons tracé le temps d'exécution moyen de chaque algorithme pour différentes configurations. Nous constatons que, pour un même niveau de profondeur et avec un nombre raisonnable d'itérations, le NRPA est généralement plus rapide que le NMCS. Par exemple, jusqu'à 100 itérations, le NRPA de niveau 1 s'avère plus rapide que le NMCS de niveau 1, malgré une performance légèrement inférieure en termes de qualité de solutions. Ainsi, le NMCS de niveau 1 peut être plus performant qu'un NRPA de niveau 1, mais il est aussi beaucoup plus coûteux en termes de complexité temporelle.

manière raisonnable avec le nombre d’itérations.

Enfin, nous constatons que le temps d’exécution évolue de manière exponentielle d’un niveau à un autre. Par exemple, le NRPA de niveau 1 avec 25 itérations est environ 24 fois plus rapide que le NRPA de niveau 2 avec le même nombre d’itérations. Cette observation souligne l’impact significatif du niveau de profondeur sur la complexité temporelle.

En conclusion, ces tests ont confirmé l’existence d’un véritable compromis entre la qualité de la solution et la complexité temporelle. Il est essentiel de trouver un équilibre entre ces deux aspects en fonction des contraintes et des objectifs spécifiques de chaque application. Les résultats de nos expérimentations nous ont permis de mieux comprendre les performances de chaque algorithme et de déterminer les meilleures configurations pour résoudre efficacement le jeu de morpion solitaire.

5 Améliorations possibles

Nous avons identifié plusieurs axes d’amélioration possibles pour notre projet. Tout d’abord, nous pourrions pu continuer à optimiser nos algorithmes en explorant des heuristiques plus avancées. En utilisant une heuristique plus performante, nous pourrions potentiellement réduire davantage la complexité temporelle de nos algorithmes et améliorer la qualité des solutions trouvées. Une piste intéressante à explorer serait la technique d’optimisation Range Query proposée dans [1] qui permet d’accélérer la génération de playout. Cette approche pourrait contribuer à améliorer les performances de nos algorithmes et à réduire les temps de calcul.

En outre, une autre amélioration possible serait d’implémenter le jeu dans d’autres modes, tels que 5D, 5T#, 5D#. Bien que notre projet se soit concentré sur le développement des algorithmes d’intelligence artificielle, l’ajout de ces modes de jeu supplémentaires élargirait la variété des expériences de jeu proposées aux utilisateurs. De plus, cela nous permettrait de comparer les résultats obtenus dans la version 5T avec ceux des autres modes de jeu, et ainsi évaluer les spécificités et les particularités de chaque configuration. Cela pourrait être une évolution intéressante à envisager à l’avenir pour enrichir davantage notre application.

6 Conclusion

Notre projet de développement du jeu du Morpion Solitaire avec des algorithmes d’intelligence artificielle a été un défi à la fois sur le plan de l’implémentation et de la recherche de solutions. Nous avons choisi un jeu qui n’était pas seulement simple à jouer, mais aussi complexe à développer. Nous avons mis l’accent sur la création d’une interface conviviale et ergonomique pour offrir aux utilisateurs une expérience agréable.

D’autre part, la recherche de solutions à travers l’intelligence artificielle a été confrontée à la complexité inhérente du problème. Le Morpion Solitaire est un problème NP-complet, ce qui signifie qu’il est intrinsèquement difficile de trouver l’algorithme optimal pour

résoudre le jeu. Nous aurions aimé obtenir de meilleurs résultats avec nos algorithmes, mais nous avons été confrontés à des contraintes temporelles et aux limitations de nos ressources matérielles. Une exécution plus longue ou une configuration matérielle plus puissante auraient pu potentiellement améliorer les performances.

Cependant, malgré ces défis, notre projet nous a permis d'approfondir notre compréhension du Morpion Solitaire et de ses complexités. Nous avons exploré les méthodes Monte Carlo et leur puissance dans la résolution de problèmes d'optimisation. Nous avons également mis en pratique des techniques d'optimisation, d'heuristique et de parallélisation pour améliorer les performances de nos algorithmes.

Références

- [1] Lilian Buzer and Tristan Cazenave. Playout optimization for monte-carlo search algorithms. application to morpion solitaire. In *2021 IEEE Conference on Games (CoG)*, pages 01–08. IEEE, 2021.
- [2] Tristan Cazenave. Nested monte-carlo search. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.
- [3] Tristan Cazenave. Generalized nested rollout policy adaptation. In *Monte Carlo Search : First Workshop, MCS 2020, Held in Conjunction with IJCAI 2020, Virtual Event, January 7, 2021, Proceedings 1*, pages 71–83. Springer, 2021.
- [4] Erik D Demaine, Martin L Demaine, Arthur Langerman, and Stefan Langerman. Morpion solitaire. *Theory of Computing Systems*, 39(3) :439–453, 2006.
- [5] Christopher D Rosin. Nested rollout policy adaptation for monte carlo tree search. In *Ijcai*, volume 2011, pages 649–654, 2011.