Jallais Bastien

Toussaint Jarod

Letondal Titouan

**Compte rendu SAE – Introduction à l’intelligence artificielle**

Lien du github : https://github.com/Jarodlp/SAE-IA

1. **Perceptrons multicouches**
2. **MLP - Discrimination**

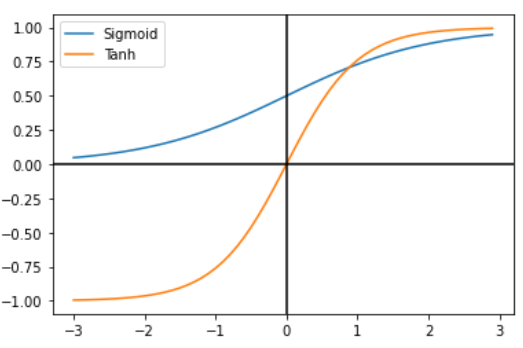
**Introduction :**

Le perceptron Multicouche (MLP) est un type de neurones artificiels largement utilisé en apprentissage supervisé pour résoudre des problèmes complexes comme la classification et la régression. Il fait partie des modèles d’intelligence artificielle inspirés du fonctionnement du cerveau humain, ou les neurones sont organisés en couches et communiquent entre eux via des connexions pondérées.

Le playground du site Tensorflow permet de simuler et de construire le comportement de perceptrons multicouches. De nombreux paramètres sont disponibles pour affiner le comportement de ces derniers.

* **Les fonctions d’activations**

Nous n’allons utiliser que les fonctions Sigmoïde et TanH. La fonction Sigmoïde n’est jamais négative, tandis que TanH peut l’être. TanH est centrée sur 0, tandis que Sigmoïde sur 0.5.



**Les poids et biais**

Nous avons donc les deux premiers neurones x1 et x2, qui ne bougeront pas, qui sortent respectivement une séparation verticale et horizontale et bien distincte sur les poids 0 et 1.

Plus le poids de chacun est important, plus l’output ressemblera à sa forme. Si les deux poids sont égaux, alors l’output est en forme de diagonale, ce qui est la moyenne des deux output de chacun des deux neurones présents. Une image contenant texte, capture d’écran, diagramme, logiciel

Description générée automatiquement

Ici sur l’image, les poids sont tous deux à 0.5. On remarque également que si on inverse les poids, ici on a mis -0,5 de chaque côté, la sortie s’inverse :

Une image contenant capture d’écran, texte, conception

Description générée automatiquement

Ici, pour bien visualiser, j’ai coché la case « discretize output » qui colore en bleu les valeurs négatives et en orange les valeurs positives. On peut décocher cette case pour avoir une visualisation plus précise, qui affiche du blanc à 0, et plus la sortie est élevée, plus la couleur est proche du bleu/orange.

Ajoutons maintenant un premier neurone dans une première couche :

Une image contenant texte, capture d’écran, diagramme, carte

Description générée automatiquement

Si les deux premiers n’envoient rien, donc des poids de 0, ce neurone ne renvoie rien non plus.

On remarque que contrairement aux deux premiers, ce neurone dispose d’un biais.

Essayons de comprendre comment la modification de ce dernier affecte la sortie.

Si un seul neurone envoie un poids, et que ce dernier renvoie un poids du même signe, avec un biais de 0, il est simplement l’image du neurone Xn qui lui envoie le poids.

Une image contenant texte, capture d’écran

Description générée automatiquement

Si on change le biais pour le passer à +2, on constate que le neurone renvoie toujours la même image, mais en accentuant les valeurs positives à hauteur de 2 dans les coordonnées :

Une image contenant texte, capture d’écran, diagramme, bleu vert

Description générée automatiquement

Naturellement, si le biais est négatif, c’est l’inverse qui se produira.

**Taux d’apprentissage**

D’autres paramètres sont présents comme le taux d’apprentissage (learning rate)

Lors de l’apprentissage, le réseau ajuste ses poids en suivant une descente de gradient.

Cet ajustement peut être défini par le learning rate qui est un paramètre que l’on peut modifier. Ce paramètre n’a pas de bonne réponse : plus il est petit, plus les ajustements seront précis et longs, mais on peut se retrouver enfermé dans un optimum local et ne jamais atteindre la meilleure solution.

Plus il est grand, plus on oscille rapidement, de manière moins précise, mais on peut sortir des optimums locaux. Il faut donc trouver un juste milieu, voire adapter ce taux d’apprentissage en cours d’apprentissage.

**Exemple 1 – Circle**

On comprend rapidement, en lançant la résolution du problème, qu’avec cette structure on ne peut qu’obtenir une séparation linéaire.

Une image contenant capture d’écran, texte, diagramme, ligne

Description générée automatiquement

En rajoutant un neurone dans la seconde couche, on arrive à obtenir une séparation parabolique. Cependant, cette séparation ne peut pas s’isoler totalement au centre afin de correspondre à notre structure de cercle imbriqué :

Une image contenant capture d’écran, texte, diagramme, ligne

Description générée automatiquement

On dirait que chaque neurone de la seconde couche forme une ligne, et elles viennent s’annuler à leur rencontre, ce qui forme la parabole. Pour essayer de faire un cercle, on pourrait donc tenter d’ajouter une troisième ligne pour venir couper le débordement que l’on voit en haut à gauche :

Une image contenant texte, capture d’écran, diagramme, Tracé

Description générée automatiquement

Les 3 lignes forment donc une sorte de triangle délimitant les valeurs positives des négatives, et remplissant donc complètement le pattern désiré.

Pour toutes ces manipulations, j’ai utilisé la fonction d’activation TanH. Avec Sigmoïd, on a aucun mal à obtenir le même résultat avec le même nombre de neurones :

Une image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement

1. **MLP – Un peu de programmation !**

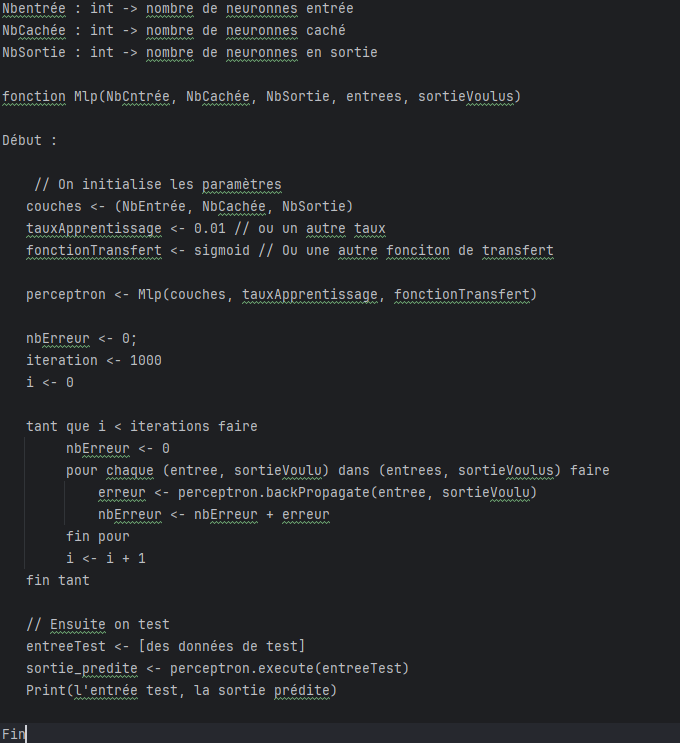
**Un squelette de code :**

Le constructeur de la classe MLP initialise le réseau de neurones avec certaines spécifications comme le nombre de neurones par couche, le taux d’apprentissage, et la fonction de transfert. Il crée les couches et les neurones en définissant les poids, biais, et valeurs initiales. Concernant le constructeur, on a en entrée un tableau (layer) indiquant le nombre de neurones pour chaque couche, un taux d’apprentissage, ainsi qu’une fonction de transfert pour chaque neurone.

La méthode execute() est utilisé pour calculer la sortie du réseau pour une entrée donnée. Elle va mettre à jour les valeurs des neurones à chaque couche, en appliquant les poids, biais, et la fonction de transfert. Cette méthode prend en entrée un tableau représentant les données en entrée qui vont être injecté dans la première couche (par exemple les caractéristiques d’une image), et renvoie un tableau (output) représentant les prédictions/résultats du réseau générés par la dernière couche (par exemple classification).

La méthode backPropagate() permet d’ajuster les poids ainsi que les biais du réseau pour réduire l’erreur entre la sortie calculée et la sortie attendue. Il utilise la rétro propagation pour propager l’erreur des couches de sortie vers les couches cachées. Cette méthode prend en entrée deux tableau input et output représentant respectivement les données d’entrée pour le réseau ainsi que les sorties attendues de ces données d’entrée.

**MLP pseudo code :**



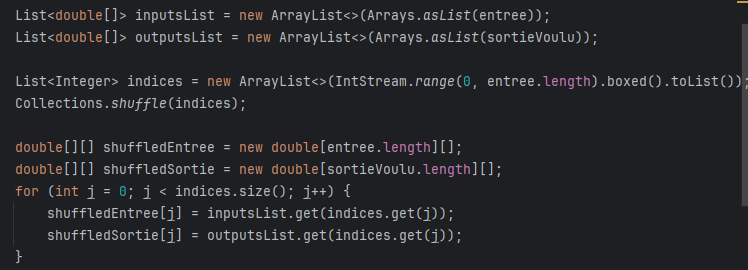
**Explication :**

* On initialise le tableau avec un nombre de neurones en entrée, dans la couche cachée et en sortie.
* Ensuite on utilise une fonction de transfert par exemple sigmoid pour dériver la classe TransferFunction
* A chaque itération de la boucle tant que, on calcule l’erreur totale en parcourant toutes les paires (entrée, sortie souhaitée) avec une boucle pour.
* La méthode backPropagate ajuste les poids et retourne l’erreur pour l’entrée courante.
* Après l’apprentissage, on teste le perceptron avec une nouvelle entrée.
* La méthode execute calcule la sortie prédite que l’on affiche ensuite.

**Code et analyse du MLP pour les problèmes AND, OR et XOR**

introduction :

Dans cette partie nous allons étudier notre programme MLP selon différents paramètres : la fonction de transfert (sigmoïde ou tangente hyperbolique), le nombre maximum d’itération, le taux d’apprentissage, et le type de problème qu’on cherche à résoudre. Pour chaque problèmes les données seront mélangées comme ceci :

Cela permet de garantir que l’entraînement est effectué de manière aléatoire et que l’ordre des exemples n’affecte pas le résultat.

Également pour ces problèmes nous considérerons une tolérance de 0,05 : par exemple si la sortie attendu est 1 et que notre MLP donne comme sortie 0,95 alors on considérera ce résultat comme juste et étant équivalent à 1.

Utilisation du code :

Lors du lancement du code, on peut chosir différents paramètres, premièrement on choisit la fonction de transfert qu’on souhaite utiliser :

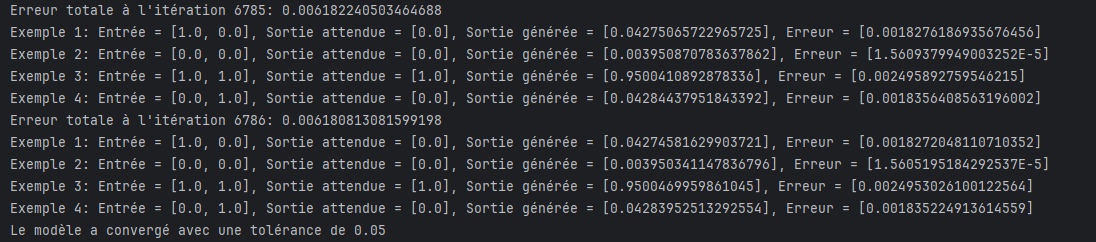
On doit ensuite choisir le taux d’apprentissage (attention il est important d’écrire par exemple 0,1 et non 0.1 :



Pour finir on choisit le problème que l’on veut résoudre :

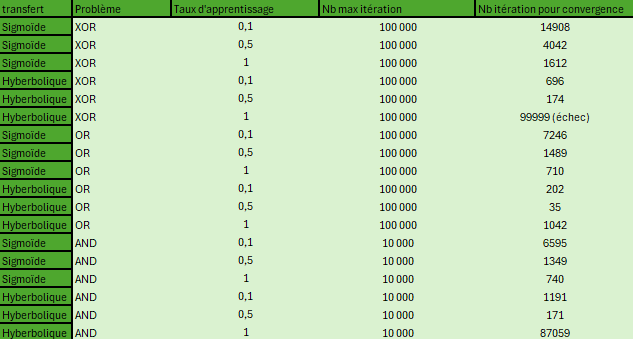


Le MLP va alors tenter de résoudre le problème en imprimant à chaque itération les entrées, la sortie attendu, la sortie générée ainsi que l’erreur restante vis à vis de cette sortie. Il s’arrêtera si une solution est trouvé ou si le nombre maximal d’itération est atteint (celui-ci peut être changé au début du code) :

Analyse

Passons maintenant à l’analyse des performances de notre MLP en modifiant les différents paramètres.

Voici un tableau qui donne le nombre d’itération nécessaire au MLP pour réussir la convergence (toujours avec une tolérance de 0,05) en fonction du problème, de la fonction de transfert et du taux d’apprentissage :

Montrer également les graphiques.

D’après ces graphiques et ce tableau, on peut tirer plusieurs conclusions notamment concernant le rôle de la fonction de transfert et celui du taux d’apprentissage sur le nombre d’itérations nécessaire à la convergence dans le perceptron multicouche.

Pour le taux d’apprentissage :

- avec un taux faible (0,1) : Les modèles semblent avoir plus de difficultés à converger rapidement, cela car car les nombres d'itérations pour la convergence sont relativement élevés, notamment pour **XOR** et **AND**.

- avec taux moyen (0,5) : En général, une augmentation du taux d'apprentissage diminue le nombre d'itérations nécessaires pour la convergence. On remarque une amélioration significative pour certains problèmes, comme pour **OR** et **AND.**

**- avec un taux élevé (1) : L'augmentation du taux d'apprentissage n'est pas toujours bénéfique. Par exemple, pour XOR, avec un taux de 1 et la fonction d'activation hyperbolique, le modèle échoue à converger (99999 itérations, soit un échec). Cependant, pour d'autres problèmes comme OR et AND, le taux élevé améliore parfois la convergence, avec des itérations plus faibles comparées à des taux plus bas.**

**Pour les fonctions d’activations :**

**- Sigmoïde : La fonction sigmoïde tend à nécessiter plus d'itérations pour converger, en particulier pour le problème XOR. Cependant, elle donne des résultats assez stables pour les problèmes OR et AND.**

**- Hyperbolique : La fonction d'activation hyperbolique converge plus rapidement pour des problèmes comme XOR et OR avec un taux d'apprentissage plus bas (0,1). Mais, comme mentionné, pour un taux d'apprentissage élevé (1), elle échoue pour XOR, indiquant une sensibilité plus élevée à des taux d'apprentissage trop importants.**

**Conclusion**

**Le taux d’apprentissage moyen (0,5) semble être un bon compromis pour la plupart des problèmes. La fonction d’activation tangente hyperbolique semble être plus efficace pour certains probmème comme OR, mais elle est plus instable avec des taux d’apprentissage trop élevés. La fonction sigmoïde est une option plus stable, bien qu’elle nécessite plus d’itération pour réussir la convergence, notamment pour XOR.**

1. **MLP vs KNN ?**

**Un squelette de code :**

1. **Conclusion**
2. **Planification & jeux**
3. **Etudier les algorithmes de planification**

**Introduction :**

Dans ce projet nous allons comparer plusieurs algorithmes de recherche de chemin de la cadre de problème de planification générés à l’aide de la classe Dummy. Voici les algorithmes que nous allons étudier :

* **BFS :** Explore tous les nœuds à une profondeur donnée avant de passer à la suivante.
* **DFS :** Explore un chemin aussi profondément que possible avant de revenir en arrière.
* **UCS :** Etend le nœud ayant le coût cumulatif le plus faible.
* **GFS :** Oriente la recherche en fonction d’une heuristique, sans considérer les coûts passés.
* **A\* :** Combine les avantages de UCS et GFS en utilisant une fonction heuristique f(n) = g(n) + h(n), où g(n) est le coût pour atteindre un nœud et h(n) est l'estimation du coût restant.

Nous analyserons les performances de ces algorithmes sur des problèmes de différentes difficultés : **Très facile**, **Facile**, **Moyen**, et **Difficile**. Chaque niveau de difficulté se distingue par la taille du graphe **n** et le facteur de branchement **k :**

A green and black text on a green background

Description automatically generated

Commençons par la difficulté la plus simple, pour celle-ci on choisit une taille de graphe n très faible (100) et un facteur de branchement également très faible (2), voici ce qu’on obtient :

A screenshot of a computer

Description automatically generated

Pour ce niveau, tous les algorithmes affichent des performances similaires, avec des temps d'exécution très courts et des coûts quasi identiques.

Cela s'explique par la simplicité du problème, où la solution est rapidement accessible, quel que soit l'algorithme. Cependant, **DFS** se distingue légèrement avec un coût sensiblement plus élevé. Cela est dû à la nature de DFS qui explore les chemins en profondeur avant de revenir en arrière, générant ainsi plus de nœuds explorés inutiles.  
En revanche, BFS, UCS et A\* affichent exactement le même coût, car ils parcourent efficacement l'espace de recherche de manière optimale

Augmentons la difficulté, voyons ce qui se passe avec un problème facile (n = 1000 et k = 3) :

A screenshot of a computer

Description automatically generated

Pour un graphe plus grand et un facteur de branchement plus élevé, les résultats restent globalement similaires. Cependant, la difficulté accrue commence à révéler certaines distinctions :

* **DFS** trouve parfois la solution plus rapidement que les autres algorithmes, bien que cela dépende fortement de l'emplacement de la solution dans le graphe. Si la solution se situe profondément dans une branche à gauche, DFS peut y accéder directement sans explorer d'autres branches.
* Le coût de DFS reste néanmoins nettement plus élevé que celui des autres algorithmes, car il génère beaucoup de nœuds inutiles lorsqu'il explore un arbre de manière non optimale.
* BFS, UCS et A\* conservent des coûts similaires, car ils partagent des caractéristiques qui garantissent une exploration optimale.

Augmentons encore la difficulté du problème :

A screenshot of a computer

Description automatically generated

**BFS** commence à afficher un coût légèrement différent de UCS et A\*, ce qui s'explique par sa stratégie d'exploration exhaustive en largeur, qui peut générer plus de nœuds que nécessaire dans certains cas.

**A**\* se montre performant, en équilibrant efficacement le coût cumulé et l'heuristique, ce qui le rend souvent plus rapide que UCS seul.

**GFS**, bien qu'orienté par une heuristique, peut parfois manquer de pertinence dans les problèmes aléatoires générés ici, ce qui explique des performances parfois inférieures.

**DFS** a trouvé une solution en seulement **0,112 secondes**, ce qui est beaucoup plus rapide que les autres algorithmes. Cela s’explique par le fonctionnement de DFS, qui explore un chemin en profondeur avant de revenir en arrière.  
Dans ce cas précis, **la solution était située en profondeur à gauche de l’arbre**, ce qui a permis à DFS de la rencontrer rapidement, par **pure chance**.

Cependant, il est important de noter que cette rapidité est aléatoire et dépend fortement de la position de la solution dans l’arbre. Dans d'autres configurations, DFS pourrait prendre beaucoup plus de temps.

Passons maintenant au problème difficile (n = 50 000 et k = 5) qui permettra de comparer efficacement les différents algorithmes :

A green and black text

Description automatically generated

Ce niveau met véritablement les algorithmes à l'épreuve. Voici les observations principales :

* **UCS et A**\* nécessitent un temps d'exécution environ deux fois plus long que certains autres algorithmes. Cela s'explique par leur recherche exhaustive et leur garantie d'optimalité. UCS, en particulier, explore souvent plus de nœuds, car il ne tient pas compte de l'heuristique.
* **DFS** devient inefficace pour ce niveau, générant un coût très élevé et nécessitant un temps parfois exponentiel en fonction de l'emplacement de la solution.
* **GFS** se montre rapide, mais au détriment de l'optimalité, car il ne prend pas en compte les coûts cumulés. Cela peut expliquer des résultats très variables selon les instances du problème.

Un point clé à noter ici est que, dans un environnement aléatoire comme celui généré par la classe Dummy, les heuristiques utilisées par A\* et GFS ne sont pas forcément très efficaces. En effet, ces heuristiques ne reflètent pas toujours fidèlement la structure sous-jacente du graphe. Cela limite la capacité de ces algorithmes à surpasser BFS ou UCS.

**Etude des résultats**

Ce graphe **montre l’évolution du temps de résolution** des algorithmes en fonction de la difficulté du problème.

A graph with lines and dots

Description automatically generated

**C**e graphe montre le **coût utilisé par chaque algorithme en fonction de la difficulté** du problème. On constate ici que UCS et A\* ont exactement le même coût peu importe la difficulté, BFS a quant à lui des coûts très similaire. On voit que GFS est légèrement plus coûteux que ces 3 algorithmes. Le plus important est de constater que DFS nécessite un coût largement supérieur et semble avoir un coût exponentiel en fonction du niveau de difficulté, ce qui le rend inutilisable pour des problèmes trop complexes.

**Que peut-on comprendre ?**

**UCS et A**\* : Ces algorithmes affichent des coûts identiques, car tous deux garantissent une solution optimale. Cependant, A\* tend à être plus rapide grâce à l'utilisation de l'heuristique.

**DFS** : Bien qu'il puisse parfois trouver une solution rapidement, son inefficacité en termes de coût le rend inadapté aux problèmes de grande taille ou aux environnements où l'optimalité est importante.

**GFS** : Rapide dans certains cas, il peut échouer à garantir une solution optimale, surtout lorsque l'heuristique manque de pertinence.

**BFS** : Efficace pour des graphes de petite taille, mais devient rapidement coûteux en termes de mémoire et de calcul pour des graphes plus grands.

**Conclusion**

En conclusion, le choix de l'algorithme dépend fortement du problème à résoudre :

* **DFS** peut être performant lorsque la solution est située en profondeur dans une branche bien définie, mais son coût élevé limite son application.
* **UCS** est adapté aux problèmes nécessitant une solution optimale, mais son temps d'exécution peut être prohibitif pour des graphes très grands.
* **A\*** se distingue par sa capacité à équilibrer optimalité et rapidité, surtout lorsque l'heuristique est bien définie.
* **GFS** est utile lorsque la rapidité prime sur l'optimalité, à condition de disposer d'une heuristique efficace.
* **BFS** reste une approche simple et efficace pour les graphes de petite taille, mais souffre de limitations dès que les dimensions augmentent.

Enfin, dans le cadre de ce projet basé sur des problèmes générés aléatoirement, l'efficacité des algorithmes est limitée par l'absence d'une structure prédéfinie ou d'une heuristique adaptée. Cela explique pourquoi les différences entre A\*, UCS et BFS restent faibles, et pourquoi DFS et GFS ne se démarquent pas significativement.

1. **Etudier les algorithmes de jeux**

**Comparons les algorithmes :**

Après avoir modifier Minmax pour ajouter de la profondeur, nous avons comparé les deux algorithmes sur plusieurs jeux, nous avons pris comme jeu le tictactoe, mnk par défaut, connect4, mnk 10x10 avec une ligne gagnante de 5 et pour finir le gomoku.

Une fois les algorithmes testés sur chaque jeu, on trouve le tableau ci-dessous, avec l’algorithme Minmax en orange et l’algorithme Alphabeta en bleu.

Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquement

Ces algorithmes ne s’affrontent pas entre eux, pour ces mesures, c’est un affrontement entre les mêmes algorithme (en orange, ce sont des mesures pour un affrontement entre deux algorithmes Minmax par exemple).

Sur ce tableau, on trouve également une erreur pour les exécutions du puissance 4, en effet l’affrontement au puissance 4 pour les deux algorithmes se solde par la victoire du joueur 2, malgré le temps accordé à essayer d’avoir la meilleure fonction d’évaluation possible pour le puissance 4.

De ces données, on peut créer deux graphiques, le premier est un histogramme pour comparer les états explorés par les deux algorithmes sur les différents jeux que vous pouvez retrouver ci-dessous.

Et le deuxième graphique ci-dessous est un graphique en courbe qui montre les états explorés par les deux algorithmes en fonction du temps.

On remarque donc, sur ces deux graphiques que l’algorithme AlphaBeta est plus performant que Minmax, que ce soit pour le nombre d’états explorés pour chaque jeu ou bien pour le temps requis pour explorés des états.

**Comparons les fonctions d'évaluation :**

Pour cette partie, nous avons coder l’algorithme de Danger (que vous pouvez retrouver dans la classe DangerPlayer.java).

Cet algorithme réagi selon les valeurs de danger qui sont sur le plateau de jeu, comme décrit dans l’article.

Pour prendre les mesures de l’algorithme Danger, on va faire des affrontements entre lui et alphabeta qui est notre algorithme le plus performant comme démontré précédemment.

Une image contenant texte, capture d’écran, nombre, Police

Description générée automatiquement

Comme précédemment, on a comparé les états explorés par jeu puis les états explorés en fonction du temps :

Comme on peut le voir, l’algorithme Danger est beaucoup plus rapide qu’AlphaBeta car il explore un nombre très limité d’états mais l’algorithme de Danger ne joue pas pour gagner mais juste pour empêcher l’autre joueur de gagner, donc dans d’autres cas que ceux testés, AlphaBeta gagne car Danger n’est pas assez performant pour gagner lui-même.