

Introduction

Le domaine des jeux vidéo est un des domaines les plus importants en termes de développement d'algorithmes d'intelligence artificielle. Il y a une course permanente à la performance de ces algorithmes dans le but d'obtenir les meilleurs scores et de dépasser l'humain dans tous types de jeux. Parmi eux, il existe Crossy Road, un jeu d'arcade très populaire, sorti en 2014 et inspiré du jeu Frogger. Le joueur y contrôle un personnage qui doit avancer de case en case en évitant des obstacles mobiles comme des voitures ou traverser des cours d'eau sur des bûches, tout cela dans un environnement dynamique et avec une difficulté croissante. Afin de concevoir une IA capable de jouer efficacement à ce jeu, il faut résoudre plusieurs défis comme la prise de décision en temps réel et l'anticipation des mouvements des obstacles.

Le but de cet état de l'art est d'analyser et comparer les différents algorithmes déjà existants pour créer un agent pouvant jouer à Crossy Road ou à des jeux similaires avec la meilleure efficacité et performance possible. Nous nous intéresserons tout d'abord aux approches basées sur des règles, puis aux algorithmes classiques de recherche de chemin et nous terminerons par les méthodes d'apprentissage automatique et par renforcement, qui sont très populaires ces dernières années dans le secteur du jeu vidéo.

En analysant les avantages et les inconvénients de chaque méthode, servira à repérer les méthodes les plus adaptées au développement d'une IA performante dans un l'optique d'obtenir les meilleurs résultats dans des parties du jeu Crossy Road. Il servira également de base pour guider le choix d'algorithmes dans le cadre de l'implémentation d'un joueur virtuel dans la version du jeu que nous avons développée.

Méthodologie de recherche

Afin de réaliser un état de l'art complet et pertinent sur les algorithmes permettant de jouer automatiquement à Crossy Road, nous avons commencé par une phase de recherche documentaire. L'objectif principal est d'identifier les travaux scientifiques, projets open source et ressources techniques les plus adaptées dans le cadre d'une future implémentation dans notre jeu.

Les outils que nous avons utilisés lors de nos recherches sont [Github](#), permettant de voir des implémentations open-source de code, ce qui nous a montré ce qui se faisait en pratique par les autres développeurs. Nous avons aussi utilisé [Google Scholar](#), qui est un moteur de recherche pour les articles scientifiques et nous a permis de découvrir des nouvelles méthodes pour résoudre ce problème. La recherche sur différents sites internet et forums spécialisés dans ce domaine a pu compléter nos connaissances sur ces algorithmes.

Analyse des algorithmes

A partir de nos connaissances et de nos recherches, nous avons pu construire une liste des principales méthodes et algorithmes habituellement utilisés pour les jeux de ce type. Lors de cette partie nous allons analyser leur fonctionnement ainsi que leurs différentes caractéristiques, telles que la performance, la complexité et le temps d'exécution. Nous avons regroupé ces méthodes en plusieurs catégories afin de permettre une comparaison plus structurée et mettre en évidence les avantages et limites à chaque type de méthode.

Systèmes à base de règles (Rule-Based AI) :

C'est une des approches les plus basique et plus simple à implémenter. Son principe repose sur un ensemble de conditions (règles) prédéfinies qui sont explicitement programmées ([référence](#)). Par exemple « Si pas de danger devant alors avancer d'une case » ou « Si obstacle devant, aller sur la case de droite ». Un des grands avantages de cette méthode est sa prévisibilité et sa fiabilité car le comportement de l'agent est entièrement déterministe et facilement contrôlable. Néanmoins, ces algorithmes présentent assez rapidement leurs limites, notamment à cause du manque de flexibilité face à des situations imprévues ou complexes. Il faut pouvoir couvrir tous les cas possibles, avec plusieurs coups d'avance, ce qui devient extrêmement complexe et inefficace dans un environnement dynamique et semi-aléatoire comme celui de Crossy Road.

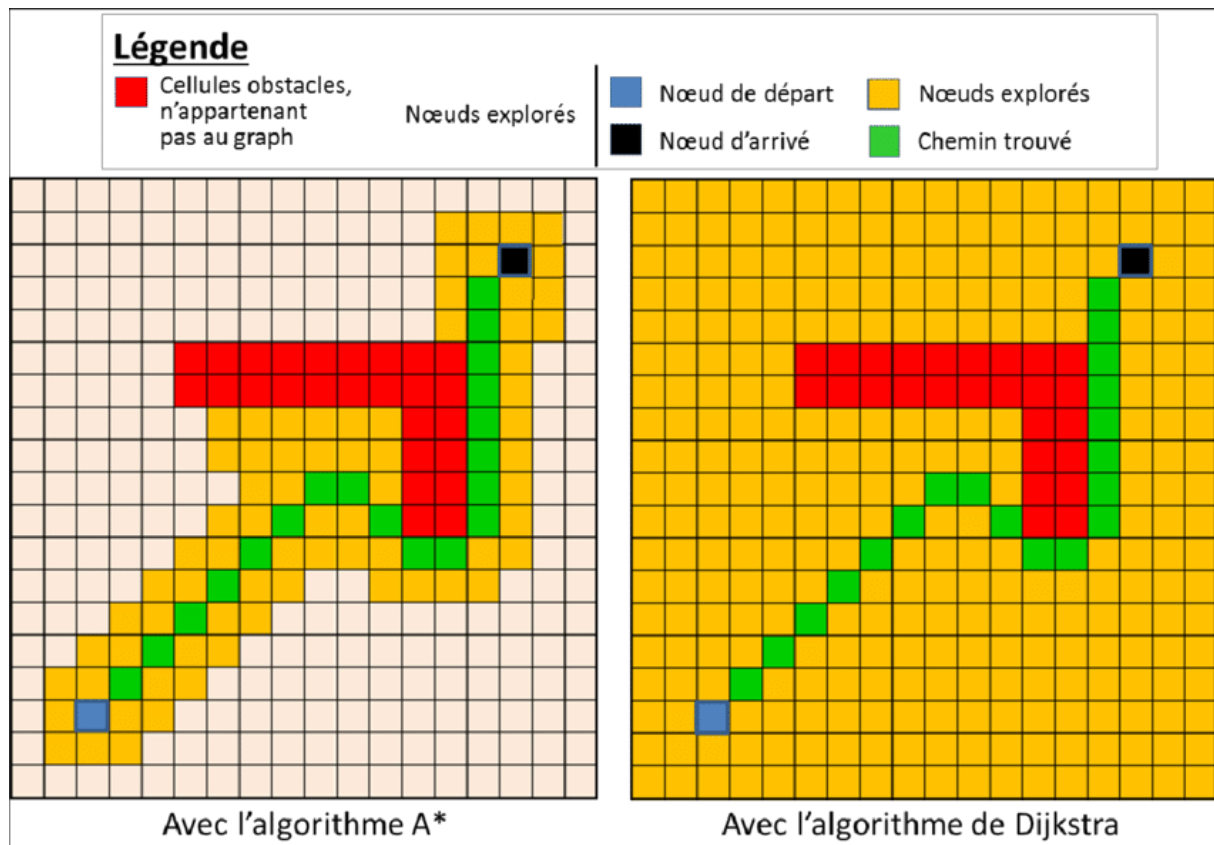
Algorithmes de pathfinding :

L'environnement du jeu peut être comparé à une grille où chaque case contient un type de surface ou un obstacle. Cela convient parfaitement aux algorithmes de pathfinding qui permettent de déterminer le chemin optimal d'un point A à un point B et donc d'avancer le plus loin possible dans le jeu en esquivant les obstacles. Ces algorithmes sont particulièrement efficaces dans des environnements connus par le joueur et permettent à l'IA d'anticiper ses déplacements en traçant un chemin qui évite les obstacles et qui est assez court pour avancer efficacement.

Il existe de nombreux algorithmes de pathfinding mais nous allons nous intéresser à deux des algorithmes les plus utilisés dans ce domaine.

[L'algorithme de Dijkstra](#) permet de déterminer le chemin le plus court entre un point a et un point b en se basant sur l'algorithme du parcours en largeur (BFS). Il va chercher itérativement le chemin le plus court vers tous les autres nœuds du graphe jusqu'à arriver à la cible. C'est un algorithme très populaire grâce à sa fiabilité. Cependant son temps d'exécution assez peut le rend difficile à implémenter pour un jeu complexe et en temps réel.

[L'algorithme A*](#) est un algorithme heuristique qui permet de trouver rapidement un chemin entre deux points s'il en existe un. C'est un des algorithmes de pathfinding les plus utilisés dans les jeux vidéo. Il combine le BFS et de l'algorithme de Dijkstra en ajoutant une fonction heuristique pour guider la recherche vers la case cible. Dans le cas d'un jeu où l'on peut se déplacer dans les quatre directions, on choisit la distance de Manhattan pour remplir le rôle la fonction heuristique. Cette fonction permet d'estimer la distance restante à chaque étape, ce qui améliore grandement la rapidité de l'algorithme, qui ne calcule pas tous les chemins possibles, mais ne garantit pas forcément le chemin le plus court.



Comparaison des nœuds explorés entre les algorithmes de Dijkstra et A*

Source de l'image : https://www.researchgate.net/figure/Nombre-de-noeuds-explores-entre-lalgorithme-A-et-lalgorithme-de-Dijkstra-Les-deux_fig30_318115999

Apprentissage par renforcement :





L'apprentissage par renforcement consiste à entraîner un agent à maximiser son score en interagissant de nombreuses fois avec l'environnement. L'agent apprend et s'améliore à chaque tentative en recevant des récompenses ou des malus en fonction des conséquences de ses actions. Ainsi, l'IA développe elle-même sa stratégie au fil du temps et peut la réutiliser sur des environnements qu'elle n'a pas encore vus. C'est le cas de l'environnement de Crossy Road avec sa génération procédurale, différent à chaque partie.

Cette catégorie est aujourd'hui l'une des méthodes les plus efficace pour obtenir les meilleurs résultats, surpassant les humains, dans les jeux vidéo et jeux de stratégie. On peut prendre l'exemple du programme [AlphaZero](#) de DeepMind, qui avec seulement 24 heures d'entraînement au jeu d'échecs a obtenu un niveau de jeu supérieur aux humains et a aussi battu les programmes champions du monde de l'époque.

Appliqué à un jeu comme Crossy Road, le renforcement permettrait à un agent d'apprendre à survivre et progresser de manière autonome, en adaptant ses décisions aux différents types d'obstacles et de situations rencontrées.

Il y a deux algorithmes d'apprentissage par renforcement qui sont principalement utilisés dans les jeux tels que Crossy Road : le Q-Learning et le Deep Q-Network (DQN).

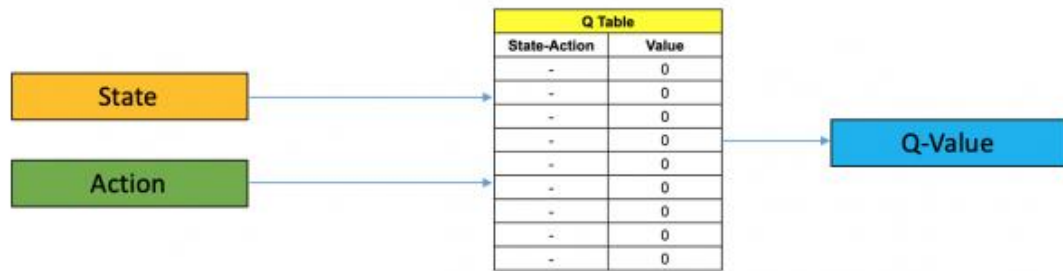
Le Q-Learning est un algorithme basé sur une table de valeurs appelée Q-table. Chaque case représente un état de l'environnement et une action de l'agent et contient une valeur qui représente l'utilité de cette action pour maximiser le score. Après chaque essai, l'agent met à jour cette table, en apprenant quelles actions maximisent la récompense cumulée. Cela est efficace dans des environnements simples ou de petite taille, qui comportent peu d'états, cependant le Q-Learning devient vite limité dès que l'espace des états est trop grand, comme c'est le cas dans Crossy Road, si l'on veut anticiper les mouvements en prenant en compte les cases éloignées du joueur.

| |  |  |  |  |
|-------|---|---|---|---|
| Start | 0 | 1 | 0 | 0 |
| Idle | 2 | 0 | 0 | 3 |
| Hole | 0 | 2 | 0 | 0 |
| End | 1 | 0 | 0 | 0 |

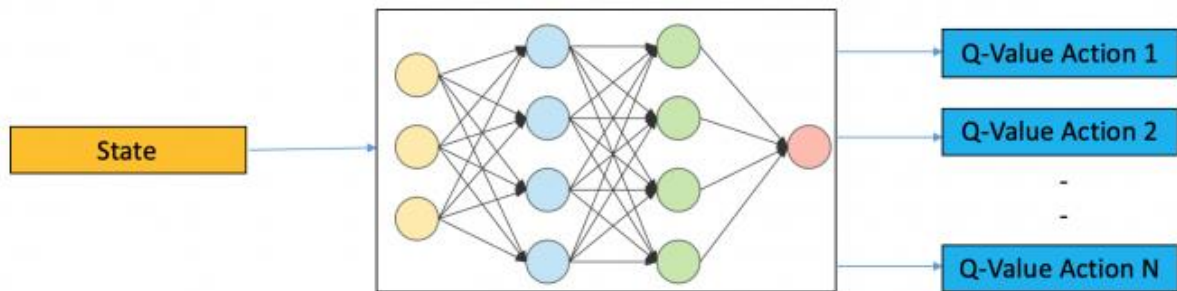
Exemple Q-table avec état-déplacements

Source de l'image : <https://www.datacamp.com/fr/tutorial/introduction-q-learning-beginner-tutorial>

Le Deep Q-Network (DQN) est une extension du Q-Learning qui remplace la Q-table par un réseau de neurones. Ce réseau prend en entrée les caractéristiques de l'état de l'environnement et prédit la valeur. Cela permet à l'agent de généraliser à partir de ses expériences passées, sans avoir besoin de stocker toutes les valeurs état-action dans une table. Grâce à cette approche, le DQN peut gérer des environnements beaucoup plus vastes et complexes. Dans un jeu comme *Crossy Road*, où l'agent doit prendre des décisions en fonction de nombreux éléments dynamiques (voitures, troncs, rivières...), le DQN permet d'apprendre des comportements efficaces en analysant les conséquences de ses choix et en adaptant sa stratégie pour maximiser sa survie et son score.



Q Learning



Deep Q Learning

Comparaison Q Learning et Deep Q Learning

Source de l'image : <https://blog.mlq.ai/deep-reinforcement-learning-q-learning/>

Algorithmes de recherche arborescente :

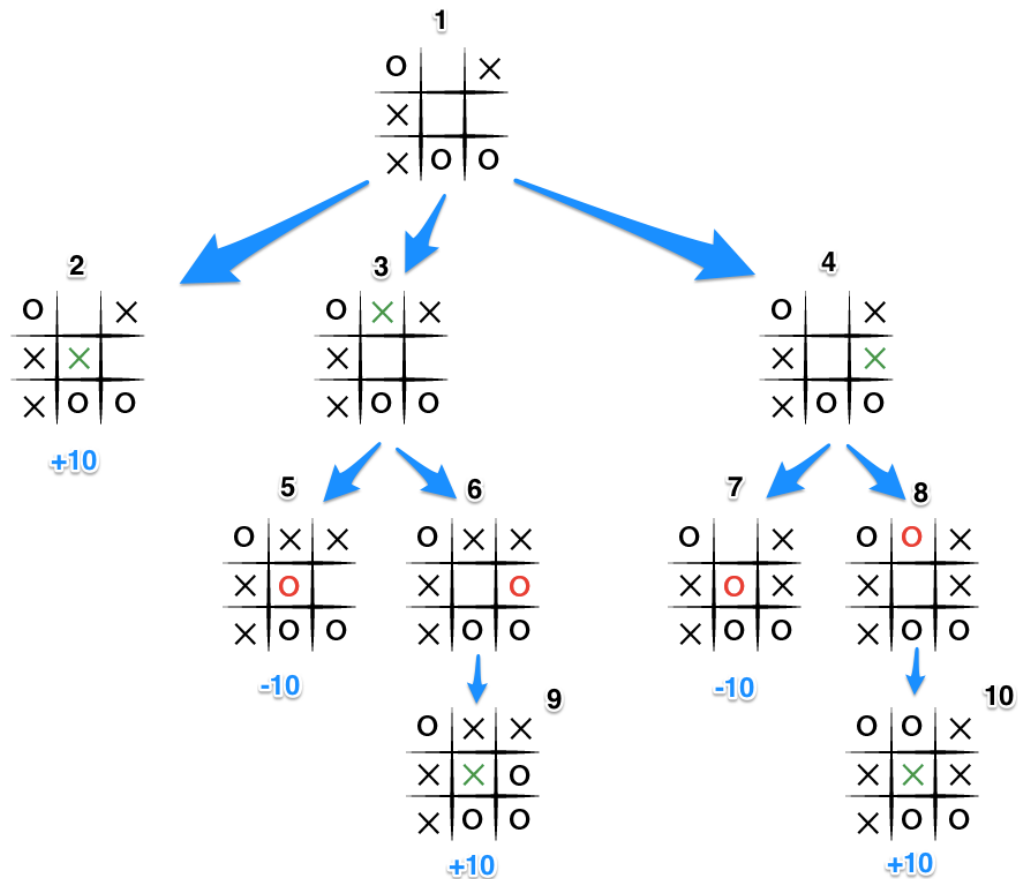
L'[algorithme Minimax](#) est utilisé dans les jeux à 2 joueurs, tels que les échecs, le morpion ou le puissance 4. Son principe est de toujours trouver le coup qui maximise le score du joueur (MAX), sachant que l'adversaire choisira toujours le coup qui minimise le score de notre joueur (MIN).

Pour cela, l'algorithme explore récursivement tous les coups possibles jusqu'à une certaine profondeur ou jusqu'à la fin du jeu. On attribue une note à chaque coup, indiquant à quel point il est gagnant pour MAX. Ensuite quand MAX joue, il choisit le coup ayant la meilleure évaluation possible. Puis quand MIN joue, il choisit le coup ayant la pire évaluation possible pour MAX. À la fin de la récursion, l'algorithme remonte les scores calculés jusqu'à la racine et renvoie le meilleur coup possible. Cet algorithme garantit une stratégie optimale si les deux joueurs jouent parfaitement.

Dans notre jeu Crossy Road, l'environnement constitue une forme d'adversaire indirect pour le joueur. Les déplacements des éléments mobiles, comme les véhicules, représentent les coups de cet environnement. Bien qu'il ne cherche pas à faire perdre le joueur, l'environnement évolue de manière déterministe ou selon des règles préétablies. Cela signifie qu'il ne réagit pas aux actions du joueur, mais son fonctionnement peut tout de même engendrer des situations dangereuses. L'algorithme est donc applicable à notre jeu si l'on considère que l'adversaire est l'environnement et qu'il ne possède qu'un seul coup qui est de faire avancer les véhicules.

L'[élagage alpha-bêta](#) est une optimisation de l'algorithme Minimax. Il permet de réduire le nombre de nœuds évalués dans l'arbre de recherche en éliminant les branches qui ne peuvent pas influencer le

résultat final. On a alpha qui représente la meilleure valeur que MAX peut garantir jusqu'à présent et bêta qui représente la meilleure valeur que MIN peut garantir jusqu'à présent. Si à un moment donné $\alpha \geq \beta$, cela signifie qu'on peut ignorer cette branche car elle ne sera jamais choisie. Cela réduit le nombre de branches à explorer et rend la recherche plus rapide sans modifier le résultat attendu.



Algorithme minimax appliqué au jeu du morpion

Source de l'image : <https://www.neverstopbuilding.com/blog/minimax>

Conclusion

Grâce à cet état de l'art, nous avons pu analyser différentes méthodes et algorithmes permettant à une IA de jouer au jeu Crossy Road. Chaque méthode possède ses avantages et inconvénients, tels que la qualité des résultats, la complexité algorithmique, ou encore la difficulté d'implémentation.

Ainsi, les systèmes à base de règles sont simples à implémenter et à comprendre, mais leur manque d'adaptabilité les rend peu efficaces dans des environnements complexes. Les algorithmes de pathfinding comme Dijkstra ou A* sont adaptés à des environnements où l'on peut trouver un chemin d'un point A à un point B mais doivent être adaptés lorsque l'état du jeu évolue, comme avec des déplacements d'obstacles. Enfin, les méthodes d'apprentissage par renforcement, comme le Q-Learning ou le Deep Q-Network, sont de plus en plus utilisées et semblent être les plus efficaces

grâce à leur adaptation aux différentes situations rencontrées. Cependant l'entraînement de ces modèles peuvent se révéler compliqués à mettre en place et nécessiter une grande puissance de calcul. L'algorithme proposant le meilleur compromis entre efficacité, complexité d'implémentation et temps de calcul semble être l'algorithme A* qui est populaire dans ce type de jeu.