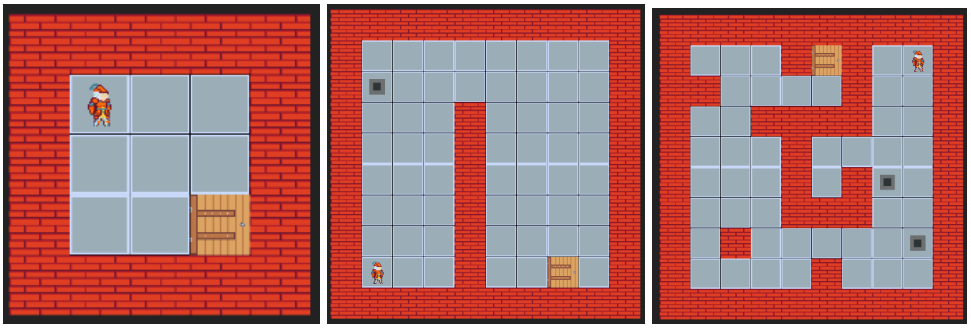
Analyse des résultats de Deep Reinforcement Learning

Les algorithmes présentés ici donnent un bon résultat pour Gridworld ainsi que pour Sokoban dans un cas avec une caisse. Nous parvenons également à obtenir un résultat fonctionnel avec 2 caisses mais ce n'est pas le chemin le plus efficace.
Pour obtenir le temps moyen d'exécution de nos algorithmes, nous lançons 100 itérations.

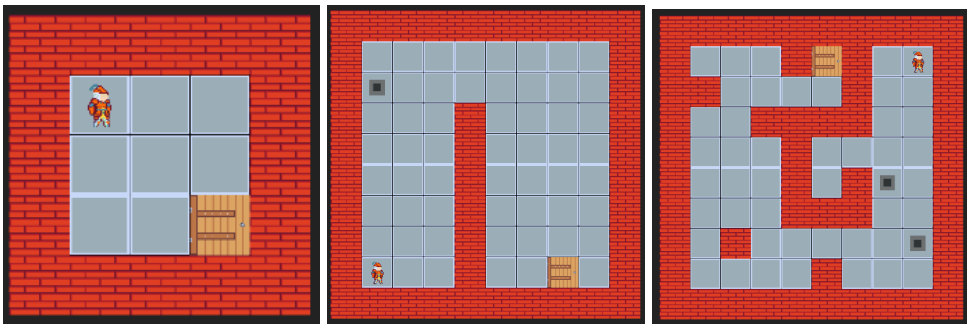
Analyse de Gridworld :
Résultat de Value Iteration :
Pour une grille de 9 cases parcourables, l'algorithme met en moyenne 0.0002004 secondes pour trouver un résultat correct.
Pour une grille de 58 cases parcourables, l'algorithme met en moyenne 0.0091157 secondes pour trouver un résultat correct.
Dans les 2 cas, le temps d'exécution est dérisoire.



0.0268797 0.3797084 0.2678055
Temps d'exécution sur 100 itérations

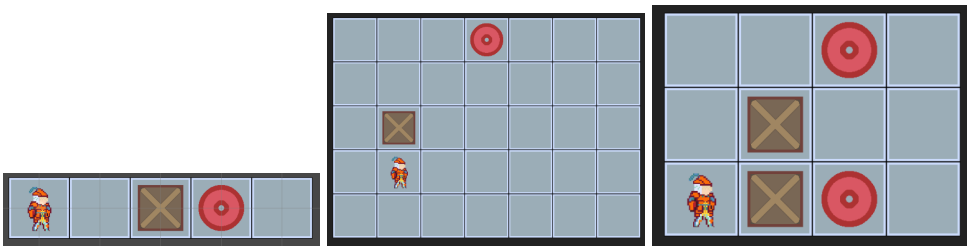
Résultat de Policy Iteration:
Pour une grille de 9 cases parcourables, l'algorithme met en moyenne 0.0008674 secondes pour trouver un résultat correct .
Pour une grille de 58 cases parcourables, l'algorithme met en moyenne 0.0272555 secondes pour trouver un résultat correct .
On obtient ici aussi des valeurs très basses même si elles sont sensiblement supérieures aux résultats de Value Iteration.

On peut en déduire que les algorithmes de Value et Policy Iteration sont plutôt adaptés à Gridwork. La seule information d'un état à traiter étant la position du joueur, on peut se permettre de parcourir tous les états.



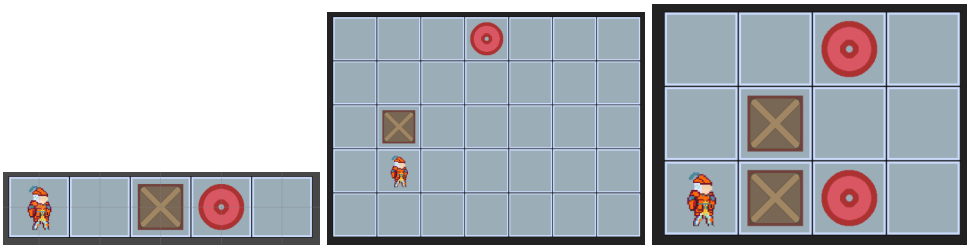
0.0885516 0.7977044 0.9074080
Temps d'exécution chaque test on été fait sur 100 itérations

Analyse de Sokoban :
Résultat de Value Iteration :
Pour une grille de 35 cases parcourables et 1 caisse à placer, l'algorithme met en moyenne 1.7113849 secondes pour trouver un résultat correct .
Pour une grille de 12 cases parcourables et 2 caisses à placer, l'algorithme met en moyenne 2.3365636 secondes pour trouver un résultat correct.



0.2020747 03:00.2604532 03:50.6346053
Temps d'exécution sur 100 itérations

Résultat de Policy Iteration :
Pour une grille de 35 cases parcourables et 1 caisse à placer, l'algorithme met en moyenne 2.4615584 secondes pour trouver un résultat correct .
Pour une grille de 12 cases parcourables et 2 caisses à placer, l'algorithme met en moyenne 5.1680829 secondes pour trouver un résultat correct.



0.1817553 04:13.9368177 08:13.3930979
Temps d'exécution sur 100 itérations

Le temps d'exécution des deux algorithmes est beaucoup plus lent que sur Gridworld. Cela s'explique par la complexité qui augmente exponentiellement en fonction du nombre de caisses dans le niveau. En effet dans Gridworld il y a un état par position du joueur possible, tandis que dans Sokoban on ajoute également un état par position de caisse possible pour chaque position du joueur. Puis on refait le même traitement pour chaque caisse supplémentaire.
On remarque aussi que Policy Iteration est bien moins efficace que Value Iteration. De plus, en regardant les valeurs minimales et maximales de nos 100 itérations sur Policy Iteration, on remarque que le temps d'exécution n'est pas du tout fixe : celui-ci varie entre 3.3638329 et 8.0002294 secondes alors que Value Iteration ne varie qu'entre 2.1596237 et 4.4410130 secondes

Structure du projet

Afin de limiter la quantité de données à calculer pendant l'exécution des algorithmes, nous effectuons certains traitements au préalable. Tout d'abord, nos états stockent uniquement les informations essentielles à leur traitement. Ainsi dans le cas de Gridworld, le seul élément qui différencie un état d'un autre est la position du joueur. Dans le cas de Sokoban on stocke la position de toutes les caisses en plus de celle du joueur. Cela nous évite d'avoir à stocker l'entièreté de la grille dans chaque état.
Les états stockent également les actions possibles depuis la position du joueur liés à cet état. Ainsi lorsqu'on doit parcourir les actions pour obtenir la plus optimale dans Value Iteration par exemple, on limite le nombre de boucles en ayant éliminé un certain nombre d'actions qui auraient été de toute façon impossible à réaliser.
La valeur de reward est également calculée lors de la création d'un état. On peut se permettre ce calcul préalable car nous sommes dans un exemple simple et déterministe où l'on sait avec certitude dans quel état une action va nous mener depuis un état de départ

Nous avons rendu les algorithmes aussi génériques que possible pour qu'ils ne dépendent pas du jeu sélectionné. Ils ne nécessitent qu'une référence vers le GridController qui contient les informations de la partie puis se déroule de façon identique pour Gridworld et Sokoban.

Axes d'amélioration :

- Optimisation : Passer certaines classes comme la classe State en Struct
- Ajouter les algorithmes MCES, Sarsa, Q-learning
- Refactor : la structure globale est plutôt brouillonne. Sur la fin du projet, nous avons passé certaines variables ou fonctions en public pour nous concentrer sur la correction des algorithmes. Certaines parties de codes se répètent comme par exemple le switch qui lance l'algorithme choisi. Certaines fonctions ont été ajoutées dans des classes qui ne sont pas les plus pertinentes. La fonction GetNumberOfCorrectCrates permet de calculer la reward d'un état dans Sokoban et se trouve dans la classe GridController alors qu'elle devrait se trouver dans la classe State.

Comme nous l'avons dit, dans l'état actuel du projet, la policy de Sokoban est fonctionnelle mais pas optimale. Nous avons tenté d'ajouter une reward de mi parcours en calculant le reward d'un état en fonction du nombre de caisses correctement positionnées.
Cependant cela ne corrige pas encore totalement l'algorithme. L'IA parvient à réaliser à placer une première caisse de façon plus efficace et commence à pousser la seconde mais s'arrête en mi chemin.

État final de Sokoban avec des rewards de mi parcours

Le rendu finale contient donc la version du projet avec uniquement une reward sur l'état finale qui parvient à mieux finir le chemin.

Annexe :

Configuration de test:

- Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz 2.30 GHz
- 8 go RAM
- NVIDIA GeForce GTX 960M
- Unity 2020.3.x (Debug mode)

Data :

Level	Algo	Game	Min Time	Max Time	Arevage Time	Walkables Tiles	Nb Iteration	Crates
1	Value	Grid	0.0000001	0.0029933	0.0002004	9	100	0
2	Value	Grid	0.0079739	0.0100113	0.0091157	58	100	0
3	Value	Grid	0.0079781	0.0199513	0.0095769	48	100	0
1	Policy	Grid	0.0000001	0.0040452	0.0008674	9	100	0
2	Policy	Grid	0.0199198	0.0349074	0.0272555	58	100	0
3	Policy	Grid	0.0179845	0.0329452	0.0266757	48	100	0
1	Value	Soko	0.0009674	0.0119682	0.0021996	5	100	1
2	Value	Soko	1.6404688	2.2725509	1.7113849	35	100	1
3	Value	Soko	2.1596237	4.4410130	2.3365636	12	100	2
1	Policy	Soko	0.0009425	0.0049992	0.0015185	5	100	1
2	Policy	Soko	2.1284346	3.0476803	2.4615584	35	100	1
3	Policy	Soko	3.3638329	8.0002294	5.1680829	12	100	2