MARION Judicael
ROUIBAH Ryan

Compte Rendu APP ALGO6 : Sokoban

Lors de ces 3 semaines d'APP, notre objectif a été de créer une intelligence artificielle permettant de résoudre des niveaux du jeu Sokoban. Plusieurs problématiques se sont posées lors de ces semaines, comment peut-on modéliser notre problème ? Quelles structures de données utiliser ? Pourquoi pas aller élever des chèvres en montagne au final ? Est ce que notre algorithme fonctionne ? Dire que tout s'est pas passé comme prévu serait une euphémisme puisqu'on arrivait pas vraiment à voir quelle direction on voulait prendre dans notre modélisation, on savait qu'on voulait utiliser des graphes, et faire du plus court chemin, mais impossible d'assembler les deux bouts et d'avoir quelque chose de cohérent. Jusqu'à ce que l'on se fixe sur une modélisation de notre problème qui tient la route.

Nous avons donc décidé de modéliser notre niveau avec des états générés au fur et à mesure de l'exécution de notre algorithme, ces états sont la configuration de notre plateau, autrement dit, la position des caisses et celle du pousseur. Avec ça en main, on connaissait notre état initial (qui est tout simplement l'état initial de la grille) et l'état final, (soit toutes les caisses sont sur les buts, soit il ne trouve pas de solution) et on pouvait les modéliser. En prenant en compte notre modélisation, nous avons décidé d'implémenter un algorithme A* agissant sur nos états de nos niveaux. Ca a été un peu difficile à concevoir au départ car on avait peur d'avoir à générer tous les états, alors qu'en réalité on génère uniquement les états accessibles, Mais comment faisons nous ? C'est bien plus simple que ce que l'on imaginait puisque les états accessibles sont en fait les états voisins à notre état courant, autrement dit il existe une action permettant de passer de notre état courant vers l'état suivant. Et donc, pour générer un nouvel état, il suffit de déplacer le pousseur. On déplace donc celui-ci dans toutes les directions possibles, ici 4.

Dans A*, chaque état est ajouté dans une file a priorité. On a les états pour le moment, mais on n'a pas la priorité. C'est un souci, mais on peut y remédier assez vite en utilisant un heuristique permettant de "noter" nos configurations. L'heuristique

que nous avons décidé d'utiliser est assez basique et est calculée rapidement au début de l'exécution de notre IA. Notre grille est parcourue de gauche à droite et de haut en bas afin de trouver les buts dans celle-ci, puis on parcourt la grille une seconde fois afin de calculer la distance minimale entre chaque case et les buts précédemment trouvés, on stocke ces données dans un tableau, celles ci seront utiles lors du calcul de notre priorité puisque notre heuristique est la somme des distances minimales entre chaque caisse et chaque but. On cherchera au fur et à mesure de l'exécution d'A* de le minimiser en déplaçant les caisses vers les buts.

Malheureusement, toute cette génération va créer beaucoup d'états identiques donc pour cela il faut les cataloguer. On a tout d'abord utilisé un tableau dynamique, naïvement, enfin pas tant naïvement que ça en réalité parce que initialement on voulait utiliser une table de hachage, mais qu'on avait aucune idée de comment hacher notre structure de données, donc on s'est un peu rabattus sur un tableau dynamique par dépit. Mais la complexité temporelle O(n) lors de la recherche s'est très vite fait ressentir sur beaucoup de niveaux, même des niveaux très petits. Comme l'objectif c'est quand même de faire une IA qui trouve une solution plus vite que nous, et on est pas les plus rapides, on a fini par ramener notre structure a une structure que Java sait hacher et on a pu finalement utiliser la table de hachage.

Notre algorithme fonctionne, notre pousseur trouve une solution pour aller au but, mais il nous faut encore extraire la séquence de coups qui a été effectuée pour la jouer. On a choisi d'ajouter à notre structure une référence à l'état précédent et le mouvement effectué pour y accéder. Quand on finit, on empile l'instruction de l'état et celles de tous ses prédécesseurs dans une pile. Enfin il suffit de dépiler les instructions et de les faire faire à notre pousseur.

```
A_star_SOKOBAN(Jeu J){
// Renvoie une pile d'instructions
        Mettre à 0 le niveau et enregistrer l'état initial dans S
        Initialiser une file a priorite FAP_S
        Initialiser une table de hachage H_S
        Initialiser le tableau de l'heuristique pour la file a priorité
        FAP_S.Ajouter(S)
        H_S.Ajoute(S,faux)
        Tant que FAP_S n'est pas vide faire :
            |       S=FAP_S.Retirer()
            |       H_S.Modifie(S,vrai)
            |       Si S correspond à l'état final:
            |           |       On crée puis renvoie la pile des instructions
            |        Sinon:
            |           |       Pour chaque direction D faire:
            |           |               |       S'= L'état correspondant au pousseur avançant
```

```
            |       |           |           dans la direction D
            |       |           |       Si H_S ne contient pas comme clé S':
            |       |           |           |     FAP_S.Ajouter(S')
            |       |           |           |     H_S.Ajoute(S',faux)
            |       |           |       Fin de Si
            |       |     Fin de Pour
            |       Fin de Si
        Fin de Tant que
        Renvoie une pile vide, pas de solutions
}
```

Pseudocode de notre IA (version finale)

On a enfin notre séquence de coups permettant au pousseur de finir le niveau, mais il reste encore beaucoup d'optimisations à faire. Par exemple: sur des très grands espaces, notre pousseur s'amuse à dribbler avec la caisse, ce qui est rigolo a voir, mais pas très efficace. On génère aussi beaucoup d'états, on nous a suggéré d'éliminer le pousseur de nos états, réduisant grandement le nombre d'états, mais on a pas vraiment eu le temps de se pencher sur cette éventualité. On pense aussi que notre heuristique seule n'est pas suffisamment efficace, que notre priorité a besoin de paramètres supplémentaires, on avait pensé à privilégier les déplacements de nos caisses mais des fois le pousseur préférait ramener la caisse loin des buts que l'en rapprocher.