IA para Sokoban 

Trabalho de Ariana Gonçalves 89194 e Diogo Correia 90327

Introdução à Inteligência Artificial - 2020/21

Universidade de Aveiro

Algoritmo

A cada nível a classe **Agent** é gerada e recebe o novo mapa convertendo-o num mapa de **PathFindingNode**'s. Esta é importante pois, efetua a ligação entre as árvores geradas no **TreeSearch** do keeper e das caixas. Ou seja, a cada movimento da caixa, existe um movimento do keeper associado.

Agente:

- conversão do *mapa* em nós do tipo **PathFindingNode;**
- criação de um **GameStateNode** *root* com o mapa inicial, e de um **GameStateNode** *goal* com o mapa que é expectável no final (o mapa onde as *caixas* já estão nos *goals*);
- através da *root* e do *goal* é criado uma **TreeSearch** que fornece o **path** (caminho das *caixas* aos *goals)*, do tipo **GameStateNode;**
- o **path** do keeper tambem é determinado pela **TreeSearch** e é do tipo **PathFindingNode**, que posteriormente é transformado numa string de teclas para enviar para a função *solver* do **student.py.**

Algoritmo

Pesquisa:

- representada pela classe **TreeSearch**, é pesquisada uma lista de nós (do tipo **PathFindingNode** ou **GameStateNode**) através da função **search**, lista essa que representa o caminho ótimo de um nó *start* a um nó *goal*, consoante a estratégia escolhida. Verificamos que a melhor estratégia foi a "*greedy*" tendo em conta o código atual;
- a função **children** tem um papel importante, isto porque efetua a filtragem dos nós irrelevantes aquando da nossa pesquisa. Nós como paredes, *pushes* impossíveis de ser realizados e *deadlocks* são descartados antes da *search*.

Nós:

- foram criados 2 tipos de nós : **PathFindingNode** e **GameStateNode**;
- o problema foi dividido em 2 árvores (*search*), sendo uma atribuida aos movimentos feitos no espaço físico e a outra ao estado de jogo (respetivamente);
- o **PathFindingNode** fica responsável pela movimentação do *keeper* e o **GameStateNode** responsável pela movimentação das *caixas*. No entanto, ambas comunicam entre si (estão interligadas) através de funções de verificação de existência de *path* e da interação do *keeper* com as *caixas*.

Algoritmo

Solver:

- no ficheiro **student.py** existe uma função **solver** que vai desencadear a ação do **Agente**, criando um novo objeto a cada iteração, atualizando assim também o domínio do problema. O **Agente** vai debitar as *keys* para uma lista através da função **key**, lista essa que vai ser usada na função **agent_loop** que vai enviar a key para o servidor.

Deadlocks:

- cada nó do tipo **PathFindingNode** tem uma função **is_deadlock** que verifica se esse mesmo nó representa ou não um *deadlock* no contexto do jogo do Sokoban;
- a verificação é feita através da criação de um objeto do tipo **DeadlockAgent** que, quando chamada a função **check_all_deadlocks**, vai devolver *True* se alguma das funções de um tipo específico de *deadlock* for também *True*, ou False se forem todas *False*.

Resultados

O **Agente** consegue encontrar solução para **67** *puzzles* até crashar no **68º**.

Poderá, muito provavelmente, encontrar solução para mais *puzzles*, mas como estão para lá do **68º**, nunca são testados.

As soluções para os *puzzles* são encontradas com relativa rapidez, exceto em meia dúzia de *puzzles* a partir do **40**.

Conclusões

Pontos positivos:

- o trabalho ficou bem estruturado em classes;
- momentos de boa reutilização de código;
- várias funções bastante compactas (por exemplo, uso de *list comprehensions*).

Pontos negativos:

- deteção de *deadlocks* possivelmente incompleta;
- heurísticas usadas podem não corresponder às mais eficientes;
- possível existência de estados repetidos, que levam a um maior custo de processamento.