Development of an autonomous agent for the game Rush Hour

Marco Almeida (103440)

Rui Machado (65081)

Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática

Arquitetura – Search Domain

O nosso agente está dividido em 3 componentes que são Search Tree, Domain, e Student. O Search Tree e Domain são usados em conjunto para calcular a solução dos níveis apresentados. O Student é o componente que está conectado ao servidor e efetua as soluções encontradas. Existe ainda um ficheiro externo commonplus.py que contém funções úteis aos vários componentes.

- **Search Domain**

Este componente serve principalmente para, em cada estado da pesquisa, verificar quais os movimentos possíveis e retornar os estados resultantes desses movimentos. Além disso é neste componente que implementamos as funções de custo e heurística.

- **Custo**

A função de custo implementada retorna a quantidade de movimentos do cursor da próxima ação + o custo acumulado.

Desta forma conseguimos que a solução encontrada seja a solução mais eficiente no contexto do nosso problema, já que quanto menor for o número de movimentos de cursor maior será a pontuação final.

Para implementar esta função tivemos de adicionar um atributo “cursor” ao SearchNode, para poder calcular a diferença entre a posição final do cursor no estado anterior e a posição do cursor no fim da ação presente.

Arquitetura – Search Domain - Heurísticas

- **Heurísticas -** Para a implementação de heurísticas, pesquisamos exaustivamente sobre este tópico e tentámos implementar uma série de heurísticas, sendo o objetivo principal reduzir o número de nós expandidos na pesquisa da solução e consequentemente, a reduzir do tempo demorado.

1. Definimos uma heurística que retorna a distância desde o carro vermelho até à saída. Esta heurística é admissível uma vez que o número de ações que precisamos de fazer para chegar ao objetivo é, no mínimo, a distância do carro vermelho à saída. Assim, esta heurística não sobrestima o número real de ações e, portanto, é admissível (Não aplicável ao nosso objetivo de pesquisa referenciado no próximo slide).
2. Foi implementada outra heurística que retorna o número de carros a bloquear o caminho para a saída. Esta heurística é admissível porque o número de movimentos necessários para desbloquear o carro vermelho é, no mínimo, igual ao número de carros a bloqueá-lo.
3. Finalmente implementámos uma variação da heurística anterior em que além do algoritmo já implementado verificamos se os carros que estão a bloquear o carro vermelho também estão bloqueados em todas as direções e nesse caso adicionamos um movimento à heurística.

Testámos outras variações deste conceito, em que exploramos o número de carros a bloquear o carro vermelho ou carros a bloquear esses carros, mas chegámos à conclusão que nenhuma dessas heurísticas produzia um resultado melhor.

Depois de testar todas as heurísticas, concluímos que a 3ª é a que produz os melhores resultados. O impacto da heurística é particularmente relevante nos níveis maiores (7*7, 8*8, etc).

Arquitetura – Search Tree

- A Search Tree calcula a solução para os níveis ignorando o cursor (sequência de ações de jogo). É de notar que o objetivo da nossa pesquisa é encontrar um estado em que o carro vermelho não esteja bloqueado em vez do objetivo real (carro vermelho no extremo direito do mapa).
- Decidimos usar a classe Map para definir os estados. Importante notar que para a comparação de estados (testar se são iguais), decidiu-se utilizar grids em formato string (atributo grid_txt de Map) em vez de comparar objetos Map ou atributos Map.grid. Este método aumentou a performance da Search Tree em 4 a 5 vezes.
- Corremos um **profiler** para analisar que funções estavam a gastar mais tempo. Com a utilização desta ferramenta percebemos que algumas funções são chamadas milhões de vezes. Logo, otimizações nos módulos do domínio (e por extensão, as funções auxiliares do commonplus.py lá utilizadas) foram bastante importantes para diminuir o tempo de execução de cada função. Cada função foi revista e grande parte alterada para produzir um resultado melhor.
- Reconhecemos que **ainda é possível fazer melhorias** à performance e uma das formas é substituindo a classe Map por tuplos, uma vez que a criação de objetos em Python é bastante dispendiosa. Decidimos não ir por este caminho para não complicar a leitura e compreensão do código. Apenas criamos objetos quando estritamente necessário e portanto utilizamos simulações em grid_txt para determinar efetivamente quando é necessário. Este método permite-nos criar 3 vezes menos objetos Map, de acordo com o cenário utilizado no profiler.

Arquitetura - Student

- O Student é o componente que está conectado ao servidor do jogo e que traduz as soluções encontradas no Search em movimentos do cursor.
- Assim que um novo nível é apresentado ao Student, este efetua uma search() para calcular a solução.
- A solução retornada pelo SearchTree é uma sequência de tuplos que representa os movimentos a efetuar. Cada tuplo contem a peça a movimentar e a respetiva coordenada a ocupar (ex: B, (0,0),(0,1)). Esta abordagem tem algumas vantagens para lidar com o Crazy Step e replaneamento de solução que irão ser referenciadas na secção seguinte. Adicionalmente é importante lembrar que o Student tem de adicionar o movimento do carro vermelho até ao extremo direito do mapa, visto que a Search apenas calcula o caminho até ao estado em que o carro vermelho não está bloqueado.
- Após a search() devolver uma solução (sequência de ações de jogo), esta fica guardada numa lista e o agente calcula, a cada ciclo, o movimento que o cursor precisa de fazer para executar a próxima ação do jogo e envia uma mensagem para o servidor com a tecla que produz esse movimento. Uma vez concluída uma ação a mesma é removida da lista.

Replaneamento/Crazy Step

- Se não existisse o Crazy Step, apenas uma pesquisa seria necessária para passar os níveis, mas como o Crazy Step pode afetar a execução da solução encontrada é necessário um mecanismo de replaneamento.
- Como referido anteriormente, a abordagem usada na solução retornada pelo Search (peça, coordenadas a ocupar no fim da acção) tem 2 grandes vantagens para lidar com o Crazy Step.
- Em primeiro lugar, o agente pode estar constantemente a fazer checks para verificar se a próxima acção da solução já foi concretizada e passar para a acção seguinte. Por exemplo, se a próxima acção for B,(0,0) e por sorte o crazy step fez com que B já esteja nessa posição, o agente passa à acção seguinte. Este check está implementado num loop, o que significa que este skip pode acontecer mais que uma vez no mesmo ciclo. Isto permite usar o Crazy Step a favor do agente sempre que possível.
- Em segundo lugar, o agente pode corrigir movimentos do Crazy Step. Por exemplo, se tínhamos B originalmente em (2,0) e queremos mover B para (1,0), mas o crazy step entretanto moveu B para (3,0), com esta abordagem o agente não é afectado por esse crazy step. Se tivéssemos usado uma abordagem de retornar tuplos com peça e movimento (ex: B,”esquerda”), neste caso o B acabaria em (2,0) o que poderia estragar a solução encontrada.
- Com esta abordagem o agente pode ignorar o Crazy Step na sua maioria e só precisa de replanear uma solução no caso do Crazy Step bloquear um movimento (Ex: no caso anterior mover uma peça diferente de B para (1,0)).