Development of an autonomous agent for the game Rush Hour
Relatório

Universidade de Aveiro

DETI

Marco Almeida (103440)
Rui Machado (65081)

26 de novembro de 2022

Capítulo 1

Introdução

Este trabalho prático foi desenvolvido no âmbito da unidade curricular Inteligência Artificial na Licenciatura de Engenharia de Computadores e Informática. O objetivo é criar um agente artificial em *Python* capaz de resolver níveis do jogo *Rush Hour*, sem ter conhecimento prévio dos mesmos.

O Rush Hour é um jogo inventado por Nob Yoshigahara na década de 1970 e consiste num puzzle de tabuleiro com vários carros orientados na horizontal ou vertical. Cada carro só se pode mover na direção em que está orientado (carros horizontais só podem mover-se na horizontal) e não se podem mover vários carros simultaneamente. O objetivo é levar o carro vermelho até ao extremo direito do tabuleiro.

No contexto da disciplina, o agente tem acesso a um cursor que se pode mover em todas as direções dentro de uma *grid* e selecionar/mover as peças do jogo. O agente só pode executar uma ação do cursor a cada 0.1 segundos. Além disso foi adicionado o *"Crazy Step"* que é um movimento aleatório do jogo externo ao controlo do agente.

O objetivo deste projeto é implementar um agente, que além de conseguir resolver os vários níveis do Rush Hour consiga executar essa solução no contexto proposto (lidar com o cursor e lidar com o Crazy Step).

Na elaboração deste projeto decidimos priorizar a performance acima da legibilidade do código, salvo algumas exceções.

Capítulo 2

Arquitetura

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 traduz as soluções encontradas em ações do cursor.

Existe ainda um ficheiro externo *commonplus.py* derivado do *common.py* fornecido pelos docentes, mas com algumas adições, que contém funções úteis aos vários componentes.

2.1 Search Domain

O Domain tem uma estrutura similar ao ficheiro *cidades.py*, estudado nas aulas práticas.

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.

2.1.1 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.

- Começando do mais básico, 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.

- 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 carros a bloquear o carro vermelho é inferior ou igual à distância do carro vermelho à saída, logo é admissível.

- Decidimos implementar outra versão da heurística anterior em que os veículos bloqueados (que estejam em a bloquear o carro vermelho) contam como dois. Isto, porque para desbloquear estes carros, é preciso desbloquear o carro que os está a bloquear, executando pelo menos mais um movimento.Esta heurística também é admissível.

Tentámos implementar outras variações deste conceito, em que tentamos explorar o número de carros a bloquear o carro vermelho ou outros carros, mas chegámos à conclusão que nenhuma das heurísticas melhorava a performance do agente. Apesar de algumas heurísticas reduzirem

o tempo de pesquisa "greedy", a solução encontrada nunca é ideal. No caso da pesquisa a*, nenhuma das heurísticas apresenta melhorias significativas ao tempo de pesquisa.

2.1.2 Custo

À semelhança das heurísticas, tentámos várias implementações diferentes para a função de custo mas encontrámos uma que achamos ser muito superior a todas as outras.

A função de custo escolhida retorna a quantidade de movimentos do cursor da próxima ação + o custo anterior 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 fazer algumas alterações ao código na Search Tree, nomeadamente adicionar um atributo cursor ao SearchNode, para poder calcular a posição final do cursor no estado anterior.

2.2 Search Tree

A Search Tree implementada é similar em estrutura à Search Tree estudada no guião prático, mas modificada para o nosso problema em concreto em vez de generalizada para vários tipos de problemas.

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 objectivo real que é quando o carro vermelho chega ao extremo direito do mapa. Esta pequena alteração reduz o tempo de pesquisa na maioria dos níveis.

A questão inicial em relação à Search Tree foi como definir um estado. Tentamos inicialmente que cada estado fosse uma lista com todas as peças (id, coordenadas) mas rapidamente chegamos à conclusão que isso iria envolver uma maior complexidade nas funções intermediárias. Acabámos por usar a classe *Map()* do commonplus.py como estado, o que resolveu o problema referido.

Para evitar que a pesquisa passe por estados repetidos, todos os estados visitados são armazenados num *set* e só são criados *nodes* derivados de novos 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. Não foi utilizada a função *__repr__()* uma vez que implicava a execução de *nested loops* em cada comparação de estados, algo que prova ser muito impactante na performance.

Por fim, 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.

2.3 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 efectua uma search() para calcular a solução.

A solução retornada pelo SearchTree é uma sequência de tuplos que representa os movimentos a efectuar. Cada tuplo contem a peça a movimentar e a respectiva coordenada a ocupar (ex: B, (0,0),(0,1)). Esta abordagem tem algumas vantagens para lidar com o Crazy Step 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 acção do jogo. Uma vez concluida uma acção a mesma é removida da lista.

2.3.1 Crazy Step

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.

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 recalcular uma solução no caso do Crazy Step bloquear um movimento (Ex: no caso anterior mover uma peça diferente de B para (1,0)).

Capítulo 3

Testes e Resultados

Criámos um ficheiro de teste *test_ searchtree.py* para testar a performance das nossas estratégias. Este ficheiro calcula o tempo gasto na pesquisa e o numero de jogadas devolvido pela solução obtida. Estimamos o tempo total como o tempo de pesquisa + (custo da solução * 0.1 segundos).

Esta estimativa não conta com o Crazy Step, mas já nos dá dados suficientes para entender quais as melhores técnicas para o contexto do jogo.

Através dos nossos testes chegámos à conclusão que as únicas pesquisas com melhor performance pesquisa de custo uniforme e pesquisa A*.

A pesquisa em comprimento retorna sempre a solução com menor numero de jogadas, mas não toma em conta o cursor, então nalguns níveis perde tempo com movimentos excessivos do cursor.

A pesquisa em profundidade, apesar de ser extremamente rápida, devolve soluções muito ineficientes com um elevado número de jogadas, e acaba por perder muito mais tempo a executar a solução.

A pesquisa gulosa também é bastante rápida, mas na maior parte dos níveis retorna soluções um pouco inferiores às outras pesquisas.

A pesquisa de custo uniforme é a melhor, porque apesar, de ser mais demorada que as outras, calcula sempre a solução mais eficiente em termos de movimentos do cursor, o que acaba por compensar.

A pesquisa A* é semelhante à de custo uniforme, porque as heurísticas que desenvolvemos não reduzem o tempo de pesquisa, então acaba por ser uma versão inferior da pesquisa de custo uniforme.

Olhando unicamente para o resultado dos testes escolheríamos sempre a pesquisa de custo uniforme, mas na prática não é necessariamente a melhor. Como o nosso jogo conta com a existência do Crazy Step, se este se intrometer na solução do agente com uma frequência alta, pode ser mais vantajoso utilizar uma pesquisa mais rápida, mesmo que menos eficiente na sua solução de forma a executar o maior número de movimentos possíveis antes de recomeçar a próxima pesquisa. Nos testes em que corremos o jogo real, podemos observar que o resultado não varia muito entre as pesquisas de comprimento, gulosa e custo uniforme. O que ganhamos em eficiência podemos perder em tempo de pesquisa e vice-versa.

A nossa melhor pontuação é cerca de 539466 pontos e foi utilizada a pesquisa uniforme.

Bibliografia

[1] M. Fogleman. «Solving Rush Hour, the Puzzle.» (), URL: https://www.michaelfogleman.com/rush/. (acedido a 23.10.2022).

[2] Wikipedia. «Rush Hour (puzzle).» (), URL: https://en.wikipedia.org/wiki/Rush_Hour_(puzzle). (acedido a 18.10.2022).

[3] O. B. M. Itzik Oskovski Oded Valtzer. «Rush Hour.» (), URL: https://www.cs.huji.ac.il/w~ai/projects/2015/RushHour/files/report.pdf. (acedido a 21.10.2022).

[4] A. Hommadi. «AI-for-rush-hour-game.» (), URL: http://abbashommadi.github.io/AI-for-Rush-Hour-Game/. (acedido a 20.10.2022).