Relatório de projeto - sistemas inteligentes 2017/2018 Sokoban

Ruben Branco 50013 | Raul Silvestre 50023 | Diogo Ribeiro 50042

[Date]

Índice

**Não foi encontrada nenhuma entrada de índice.**

Introdução

Sokoban consiste num puzzle em que um mapa é definido por uma grelha, , no qual existe N caixas, N locais alvo e paredes. O objetivo é encontrar uma serie de movimentos que permitem um agente colocar todas as caixas na posição alvo.

Existem algumas regras e restrições, como o agente só pode empurrar e não puxar as caixas, só pode empurrar uma caixa e não duas adjacentes.

Devido à complexidade que o problema tem, com fatores de ramificação grandes torna o espaço de busca do problema rapidamente grande até para os puzzles simples. É necessário definir uma estratégia, de forma de heurísticas, que consiga restringir este espaço e eliminar os estados que resultam em *deadlocks*.

...

Formulação

## Representação de estados

Os estados foram representados a partir de uma classe *SokobanState*, que faz uso de um grafo para ter o estado visualizado.

Esse grafo é um dicionário em que as chaves são as coordenadas cartesianas de uma certa posição no mapa e o valor é um dos 7 possíveis caracteres que definem os diferentes tipos de entidades: ‘#’, ‘.’, ‘\*’, ‘o’, ‘A’, ‘@’ e ‘B’.

As coordenadas cartesianas são mapeadas a partir de um espaço em que o eixo

dos yy está invertido em relação a um espaço cartesiano normal, ou seja, os yy tem o seu quadrante positivo a baixo do eixo dos xx.

O estado tem também alguns métodos de suporte associados à classe para ajudar na navegação do grafo.

Exemplo:

Para um puzzle

###

#o#

#\*#

#A#

###

Resultaria um grafo da seguinte forma na estrutura de dados nativa de Python, dicionário

{(0,0): ‘#’, (1,0): ‘#’, (2,0): ‘#’, (0,1): ’#’, (1,1): ‘o’, (2,1): ’#’, (0,2): ‘#’, (1,2): ‘\*’, (2,2): ‘#’, (0,3): ‘#’, (1,3): ‘A’, (2,3): ‘#’, (0,4): ‘#’, (1,4): ‘#’, (2,4): ‘#’}

## Operadores

Existem 2 tipos de operadores e 4 sentidos dos mesmos definidos. É possível mover (*Move*) e empurrar (*Push*). Os quatro sentidos para onde se pode fazer estas operações são cima (*Up*), baixo (*Down*), esquerda (*Left*) e direita (*Right*).

A partir da classe de suporte *GraphUtil* e de métodos do *SokobanState*, é restringido as ações ao contexto, ou seja, através da posição do arrumador, é determinado as ações possíveis.

Heurísticas

Foram definidas 4 heurísticas que visam a formar uma estratégia onde se procura

minimizar a distância de jogador a caixas e das caixas aos objetivos e também evitar estados mortos (explicado na secção de analise).

As heurísticas surgiram incrementalmente, ou seja, cada heurística estende a sua antecessora ao utilizar o seu custo e adicionar sobre ele.

## Heurística 1

A heurística 1 procura minimizar a distância das caixas aos objetivos. Isto é conseguido a partir de distância cartesiana. A heurística é dada por:

Em que c é o número de caixas que **não** se encontram num objetivo, o é o número de objetivos.

O valor da heurística é mais baixo conforme a proximidade das caixas aos objetivos, premiando os estados em que estão mais perto ou até no objetivo já.

## Heurística 2

A heurística 2 utiliza a heurística 1 e acrescenta um novo cálculo semelhante, em que pretende aproximar os jogadores às caixas, querendo premiar estados em que o arrumador está mais perto, em média, das caixas.

O valor da heurística é dado por:

Em que c é o número total de caixas que **não** se encontram num objetivoe A é a posição do arrumador.

O valor da heurística é mais baixo conforme a proximidade das caixas aos objetivos e do arrumador às caixas, pois a progressão das caixas aos objetivos só é possível se o arrumador estiver perto delas para as empurrar, necessitando de premiar essa proximidade.

## Heurística 3

A heurística 3 expande, outra vez, a heurística 2 e faz uso dela, mas pretende resolver outro tipo de problemas no jogo do *Sokoban*, que é os estados mortos (*deadlock*). Uma jogada pode tornar uma tentativa de resolução impossível se, por exemplo, encostar um caixa num canto, porque o arrumador não pode puxar caixas, apenas empurra-las, portanto a caixa nunca poderá mover-se do canto, e se o algoritmo não retroceder, poderá nunca convergir para uma solução.

Portanto, esta heurística percorre as caixas que não estão situam-se num objetivo e verifica se estão num canto. Se estiverem, é dado um valor de 1000 como penalidade.

## Heurística 4

A heurística 4 utiliza a heurística 3 e procura outro tipo de situação *deadlock*, que é as paredes. Uma caixa encostada a uma parede pode constituir um estado morto, pois se está encostada a uma parede significa que sem uma abertura onde se pode retirar a caixa da parede, ela permanecerá lá, o que pode tornar o puzzle impossível se não se encontra nenhum objetivo nessa coluna ou perto dela.

Para combater estas situações, se uma caixa estiver encostada a uma parede, é varrida as colunas ou linhas adjacentes com um raio dado (raio utilizado é 3), dependendo se está encostada a uma parede na parte superior ou inferior do mapa ou se está encostada a uma parede numa parte lateral do mapa. Se houver objetivos dentro desse espaço, então o número de objetivos é anotado e o custo da heurística é sumado com ..