universidade de aveiro

Universidade de Aveiro

Departamento Eletrónica, Telecomunicações e Informática

Relatório Sokoban

Trabalho realizado por:

- Pedro Amaral nºmec: 93283
- Pedro Santos nºmec: 93221
- Ricardo Cruz nºmec: 93118

Descrição Algoritmo

Goal_search.py

Na classe **SearchTree**, o método search é utilizado para encontrar o caminho das caixas até aos objetivos.

Para a pesquisa da solução, utilizámos duas variantes do A* de forma a conseguirmos encontrar a melhor solução possível para níveis mais simples, sendo o tipo de pesquisa selecionada de acordo com o resultado da fórmula Comprimento*Altura*NºObjetivos dos atributos do mapa. Quando o valor desta fórmula é menor que 600 (normalmente mapas mais pequenos) é utilizada a pesquisa de custo uniforme, caso contrário é utilizada a pesquisa gulosa.

- **Pesquisa de custo uniforme:** Nesta variação do algoritmo de A*, consideramos apenas o custo acumulado , tendo assim este método de pesquisa um comportamento muito parecido com o da pesquisa em largura, o que assegura ser encontrado uma solução considerada ótima.
  - **Custo**: Utilizámos um custo baseado na fórmula de cálculo do score.
- **Pesquisa gulosa :** Esta pesquisa tem em conta é considerado apenas a heurística, ou seja, o custo acumulado é ignorado, aproximando-se assim da pesquisa em profundidade. O cálculo da heurística é tanto melhor quanto mais aproximada estiver do custo real, tornando-se assim mais admissível.
  - **Heurística**: Soma das distâncias de Manhattan de todas as caixas que não se encontram em nenhum objetivo, até ao objetivo livre mais próximo

No método search começamos por fazer chamadas às funções “definePassages” e “expandMap”, de seguida começamos a procura pela solução iterando sobre a lista *open_nodes*, abrimos um nó e caso este seja a solução do mapa termina a procura, caso contrário obtemos todos os tiles para onde o Keeper se pode mover no estado atual do mapa fazendo uma chamada ao método “getMoves” da classe agent_search, de seguida iteramos sobre todos os movimentos possíveis das caixas (4 para cada caixa). Para cada movimento obtemos a nova posição da caixa (após o movimento atual, “newBoxPos”) e a nova posição do Keeper (diretamente “atrás” da caixa de forma a conseguir empurrá-la na direção certa, “newKeeperPos”) verificando se:

- O Keeper se pode mover para nova posição.
- A nova posição da caixa não está ocupada por outra caixa.
- A nova posição da caixa não é uma posição que coloca o mapa em Deadlock estando num canto ou numa parede sem objetivos.
- A nova posição da caixa não coloca o mapa em Deadlock estando ao lado de uma caixa, bloqueando o movimento de ambas.
- A nova posição da caixa tem um túnel no tile seguinte utilizando a função “tunnel”.
- A nova posição da caixa é uma zona de passagem, movimentando a caixa 2 tiles na direção do movimento caso seja.
- O estado atual já foi visitado anteriormente e caso tenha sido, o movimento é ignorado.
- O estado atual está em situação de Deadlock devido ao posicionamento de uma ou mais caixas, esta verificação é apenas feita quando existem 5 ou mais caixas no mapa, sendo um exemplo de Deadlock 4 caixas a formar um quadrado.

No final destas verificações criamos um novo node com a informação e adicionamos à lista *open_nodes* e ordenados a lista, utilizando a função “insort_left”.

Quando a procura termina e temos o node com o mapa completo, obtemos o percurso do Keeper (teclas necessárias para completar o mapa) utilizando a função *get_keys*, desta maneira não precisamos de obter o percurso do Keeper para estados que não façam parte da solução durante a procura.

Descrição Algoritmo

Funções:

- **isWalled_Outer:** Bloqueia todos os tiles adjacentes ás paredes exteriores do mapa, verificando se não há nenhum objetivo neste conjunto de tiles.
- **isWalled_Inner:** Bloqueia todos os tiles que resultem em situação deadlock nas paredes interiores ao mapa, verificando se não há objetivos no conjunto de tiles.
- **isCornered:** Bloqueia os tiles que façam com que a caixa fique num canto.
- **deadlock_detection:** Função utilizada para detetar situações de deadlock de forma dinâmica.
- **isBoxed:** Verifica se encosta uma caixa a outra, impossibilitando o movimento de quaisquer uma das caixas.
- **tunnel:** Verifica se existem situações de túneis horizontais (parede superior e inferior) e verticais (paredes laterais). Caso se verifique o túnel, a caixa irá ser movida logo até ao fim do túnel. Temos definido ainda um outro tipo de macro em que verificámos se tem adjacente uma parede, que passe um conjunto de verificações (não tem objetivos, não tem caixas na linha de movimento e no lado adjacente, etc).
- **definePassages:** Verifica se há algum ponto do mapa que divida o mapa em duas áreas, para que, quando o keeper chegar a este ponto de divisão com uma caixa, a empurrar diretamente para a outra área do mapa, em vez de a deixar a bloquear a passagem, visto que obrigatoriamente o vai ter de fazer.
- **expandMap:** Verifica para mapas mais complexos, se há algum movimento que seja vantajoso ser efetuado em primeiro lugar, de forma a libertar mais tiles para o keeper se movimentar.

Nas verificações necessárias para o funcionamento das funções de restrição e ao longo do método search, a verificação se o tile é uma parede é feita várias vezes. Como o processo de chamar o mapa.py é muito dispendioso em termos de tempo, definimos dois arrays bidimensionais na classe **SearchTree,** "isWall" e "isBlocked" de modo a evitar chamadas adicionais ao mapa.py. Para verificar se é uma Wall, substituímos as chamadas ao mapa.py por um valor booleano guardado no array. No inicio do algoritmo com apenas uma chamada ao mapa.py definimos neste array todas as Walls (guardado com bool True). O array "isBlocked" é um caso mais especifico, onde vamos ter guardado adicionalmente possíveis situações de deadlock (guardado com bool True).

Agent_search.py

O método search do agent_search é utilizado para encontrar o caminho que o Keeper tem que percorrer de um ponto inicial até a um ponto final. Quando este é chamado no goal_search.py, o ponto inicial (initial_pos) é as coordenadas atuais do Keeper, e o ponto final (destination) é as coordenadas onde tem de estar para empurrar a caixa.

O método getMoves() do agent_search é utilizado para procurar todos os movimentos possíveis do Keeper num certo estado.

No construtor da classe **SearchTree** é criado um agent (criamos apenas um, onde lhe passamos o array "isWall") para ser utilizado ao longo dos métodos, nomeadamente no método definePassages, expandMap e get_keys. Ao utilizarmos o agente nestes métodos, obtemos o caminho que o Keeper tem de percorrer, e também os movimentos possíveis do Keeper num certo estado, campos necessários na realização destes métodos. No método search do goal_search.py para encontrarmos os movimentos das caixas até ao objetivo, utilizamos também o agent, uma vez que é necessário também saber os movimentos possíveis do keeper.

Resultados

Nos testes efetuados nos computadores pessoais, conseguimos alcançar o nível 151, não inclusive. Como foi dito na descrição do algoritmo, existem níveis considerados mais simples em que o agente consegue passar instantaneamente, e existem níveis mais complexos que demoram alguns segundos. Tiramos vantagem disto, e para níveis mais simples, utilizamos pesquisa de custo uniforme para termos uma pontuação maior. Nos níveis complexos utilizamos pesquisa gulosa de forma a obtermos uma solução mais rápida, ainda que por vezes, possa não ser uma solução tão eficaz.

Acabámos por utilizar pesquisa gulosa, sendo que não conseguimos completar o nível 146 de outra forma.

Conclusão

Neste trabalho desenvolvemos um caso prático da pesquisa em árvore. Ao longo do desenvolvimento foram testadas diferentes métodos de pesquisa em árvore e retiramos as seguintes conclusões:

1. O método em largura produz uma solução que contem o menor numero de pushes, no entanto os movimentos do keeper e a sua sequência pode não ser a melhor, levando a um maior número de tempo a colocar as caixas nos objetivos.
2. O A* é a pesquisa mais eficaz, encontrando um equilíbrio entre a eficácia dos pushes e tempo de resolução.
3. A variante do A*, pesquisa de custo uniforme, encontra soluções com pushes semelhantes ao método em largura, no entanto os movimentos do keeper e a sua sequência vão ser significativamente melhores quando comparado com a pesquisa em largura.
4. A variante do A*, pesquisa gulosa, encontra as soluções mais rapidamente, uma vez que baseia-se apenas na heurística, no entanto pode não encontrar a solução mais eficaz.

Apesar do método de pesquisa ser um dos fatores mais importantes na pesquisa em árvore, concluímos também com a resolução deste trabalho, que as restrições de forma a diminuir nós representa também um fator muito importante na pesquisa em árvore.