Sokoban AI

RESOLUÇÃO DO JOGO SOKOBAN COM ALGORITMO DE AI

REALIZADO POR: JOÃO CARVALHO(89059) E LUÍS SILVA(88888)

Introdução

- Este projeto foi realizado no âmbito da cadeira de IA e consistiu em implementar um algoritmo de resolução do jogo Sokoban.
- Inicialmente começamos por implementar um algoritmo depth-first search, depois tentamos implementar com algoritmo STRIPS, então um de nós dedicou-se a resolver o jogo usando STRIPS enquanto o outro aprimorava a pesquisa e torná-la mais eficiente.
- A seguir iremos mostrar a estrutura e alguns detalhes das implementações usadas.

Greedy algorithm

Além do ficheiro “student.py” foram usadas três classes adicionais:

1) “hash_table.py” - Utiliza uma lista que contém os nodes visitados e evita visitá-los de novo, evitando ciclos e melhorando a eficiência.
2) “move_map.py” - Utiliza um dictionary com um state(coordenadas de boxes e keeper) como key e o seu value é uma lista de moves((1,0),(-1,0),(0,1),(0,-1)) que são os moves para, a partir do state inicial, chegar este state key.
3) “deadlock_list” - Utiliza uma lista com os deadlocks do mapa. Primeiro verifica-se possíveis deadlocks em “cantos” do mapa, depois conectam-se dois deadlocks em cantos se tiverem continuamente junto a uma wall(também são deadlocks). Aumenta a eficiência pois reduz o número de states possíveis.

Iniciação de classes

A implementação é baseada num ciclo de pesquisa clássico, utiliza-se uma lista “open_nodes” com um determinado node associado à sua heurística, que se inicia com o node root e que vai contendo os nodes a visitar que não foram visitados antes.

Inicia-se, também, a “hash_table” com o node root, e o “move_map” com o node root associado, obviamente, a uma lista vazia. Além disso, ainda antes de começar a pesquisa, encontram-se os deadlocks do mapa.

Ciclo de pesquisa

Para encontrar uma solução do mapa, iniciamos o ciclo de pesquisa ao encontrar os possíveis movimentos do agente para as suas coordenadas vizinhas com a função “get_neighbours” que verifica os possíveis movimentos entre os totais (1,0),(-1,0),(0,1),(0,-1), esta filtragem é feita pela função “nei_filter” que verifica se a nova posição de agente não é uma WALL, ou, no caso de ser uma BOX, só poderá movê-la se atrás dela não estar outra BOX, uma WALL ou um DEADLOCK. Se ocorrer um destes casos o movimento não é possivel(onde está BOX pode ser também BOX_ON_GOAL).

Ciclo de pesquisa - continuação

Agora que já temos os moves possíveis entre os vizinhos do agente cria-se um novo node para cada move recorrendo à função “new_node”. A implementação do novo node baseia-se em criar uma cópia do node atual e fazer set do Tile do MAN para as coordenadas do vizinho em questão(e fazer clear_tile das coordenadas atuais) e, se for o caso de push duma box, fazer set dos Tiles da BOX(ou BOX_ON_GOAL) para as coordenadas atrás dela(back) e mudar também o Tile das coordenadas atuas da box para FLOOR(ou GOAL).

No fim de criar novo node adiciona-se este ao “move_map” e faz-se append deste move associado à lista de moves para chegar a este mesmo node.

Ciclo de pesquisa - continuação

Depois de obter a lista dos nodes children do node atual, percorre-se cada um e verifica-se se já foi visitado através da “hash_table”, se não foi, calcula-se a sua heurística(detalhada a seguir) e adiciona-se o tulplo(newnode,heuristic) ao open_nodes e ordena-se a lista para que os nodes com menor heurística sejam visitados primeiro.

Heurística

Para o cálculo da heurística foram testadas algumas implementações, aquela que se verificou mais eficiente foi a naive heuristic. Esta heurística retorna a soma da menor distância entre o agente com uma unplaced box e as distâncias entre cada box e seus goals mais próximos.

Para calcular a distância utiliza-se a distância euclidiana.

Solução Encontrada

Quando o node da solução do mapa é encontrada fazemos get da move_list associada a este. Percorremos a lista e, no move (1,0) associamos a key “d”, (-1,0) associamos a key “a”, (0,-1) associamos a key “w”, (0,1) associamos a key “s”.

Por fim é só colocar o agente a percorrer a string “keys” com todos os moves.

Pesquisa em àrvores com operadores STRIPS

É uma maneira distinta de encarar o problema e a sua pesquisa. Consiste em criar Predicados que descrevem o estado do Universo do problema e Operadores que representam as ações que se podem realizar a cada instante e o efeito destas sobre o estado atual do problema

Definição dos Predicados

Para modelar o problema utilizamos um total de 7 predicados. Estes serão atribuídas às posições correspondentes do mapa aquando da preparação do estado inicial do mesmo mapeando a existência de goals, caixas e do keeper em cada uma dela. Nesta preparação já em feita uma análise a posições que não deverão aceitar a caixa mas são complacentes com a presença do keeper(predicado: Not_Free_For_Box). Numa primeira versão do programa usámos apenas 5 Predicados pois não havia esta filtragem inicial de posições que não devem estar livres para caixas.

Definição dos Operadores

Os operadores são estruturas que vão manipular o estado atual alterando os predicados existentes. Definimos um total de 10 Operadores para conseguir acomodar todas as ações passíveis de ser realizadas com os Predicados disponíveis. De salientar que na primeira versão, que mencionamos no slide anterior, apenas eram necessários 4 Operadores para modelar todas as interações do agente com o universo.

Para que um operador possa ser aplicado deve, primeiro, verificar-se um conjunto de pré-condições e caso estas se verifiquem, será alterado o estado para refletir os efeitos negativos e positivos do resultado da ação.

Atribuição dos Predicados

set_initial_state é uma função que define de uma só vez o Estado inicial e o Estado final do problema.

Esta função ignora todas as posições que estejam marcadas como sendo paredes. Os Predicados Free e Not_Free_For_Box só podem ser atribuídos a tiles do tipo Floor e a atribuição destes depende se a deve ser permitido ao agente que mova para lá uma caixa. Esta verificação é feita com recurso às funçoes check_neighbours e check_for_goals. A primeira verifica se o tile está numa posição onde duas ou mais paredes impedirão o utilizador de mover mais a caixa(só é relevante se essa posição não for um goal) e a segunda verifica nas linhas e colunas que representam os extremos do mapa se em alguma destas existe um goal, apenas permitirá que a caixa seja movida para uma dessas linhas/colunas caso esta contenha um goal

Os restantes tipos de tile atibuição os Predicados da seguinte forma:

Predicado\Tile	BOX	BOX_ON_GOAL	MAN	Man_ON_GOAL	GOAL
Box_on	x				
Box_On_Goal		x			x
Keeper_On_Box_Allowed			x	x	
Goal		x		x	x