Usar a implementação Prolog do A*

2018/09/16

O algoritmo A* serve para resolver problemas. Basicamente, descobre a sequência de passos necessários para converter um estado inicial num estado final (ou objetivo).

O A* serve igualmente para encontrar caminhos em grafos. Nesse caso recebe os nomes de dois nós de um grafo, e determina a sequência de arcos que formam o caminho entre o primeiro e o segundo nó. Há algumas diferenças na forma como o algoritmo deve ser usado para encontrar soluções para problemas e para encontrar caminhos entre nós. Neste apontamento, as explicações dadas assumem que se usa o algoritmo para resolver problemas.

Para usar o A*, é necessário carregar o ficheiro com a sua definição, usando o predicado *ensure_loaded/1*, por exemplo:

```
:- ensure_loaded('a-star').
```

Depois, podemos usar duas interfaces: o predicado *astar/7* (interface original) ou o mais simples *astarps/3* (interface mais simples usada apenas em problemas em que se definem ações para transitar de estado, i.e., problemas que não se resumem a procurar em grafos).

Para problemas em que se definem ações de transição de estados, poderá ser mais fácil usar o predicado *astarps/3* (A-Star Problem Solving).

1 Usar o predicado *astarps/3*:

astarps(Init, Solution, Cost)

em que

Init, a representação do estado inicial, tem de ser dada ao predicado *astarps/3*;

Solution, uma lista representando a solução, é determinada pelo *astarps/3*; e

Cost, o custo dessa solução, é também determinado pelo *astarps/3*.

Para se poder usar o predicado *astarps/3*, são necessários os seguintes passos prévios:

Definir, em Prolog, a ação (ou as ações) de transição de estado. Cada ação de transição de estado é uma ação que recebe um estado e devolve o novo estado a que resulta da aplicação dessa ação no estado recebido. Os dois primeiros argumentos de uma ação de transição de estado são obrigatoriamente o estado recebido e o estado produzido. Além destes dois argumentos, qualquer ação de transição de estado poderá ter mais argumentos.

No caso do mundo dos blocos, a ação consiste em transportar um bloco de uma posição para outra. O primeiro argumento da ação é o estado antes de a ação ser executada, o segundo argumento é o estado que resulta se a ação for executada. Mas há outros argumentos: o bloco que será transportado, o local onde ele está colocado originalmente, e o local onde ele será colocado pela ação.

Para exemplificar representaremos um bloco por uma letra (e.g., a, b, c), um espaço por um número inteiro (e.g., 1, 2, 3) e estado por uma lista de pares (bloco, posição) que representa as posições dos blocos nesse estado. Neste mundo, um bloco pode ocupar um espaço ou estar colocado em cima de outro bloco. Além da representação dos estados, os quais são alterados pela execução de ações, é necessário descrever aquilo que é sempre verdade. Neste nosso exemplo, é preciso dizer que blocos, que espaços e que posições existem:

```
% a, b e c são blocos
block(a).
block(b).
block(c).

% 1, 2, 3 e 4 são espaços
space(1).
space(2).
space(3).
space(4).

% Se X é um bloco, então X é uma posição onde outro bloco pode ser colocado
% Se X é um espaço, então X é uma posição
position(X):- block(X).
position(X):- space(X).
```

Usamos o predicado *block/1* (*block* com um argumento) para enumerar os blocos. Usámos os predicados *space/1* e *position/1* para enumerar os espaços e as posições onde os blocos podem ser colocados.

Seguidamente mostra-se a definição em Prolog da ação move(IState, DState, Block, From, To):

```
move(IState, DState, Block, From, To):-
    block(Block),
    position(From), position(To),
    From \= To,
    member((Block, From), IState),
    \+ member((_, To), IState), % Não está nenhum bloco na posição To
    \+ member((_, Block), IState),
    replace((Block, From), (Block, To), IState, DState).
```

\= é um predicado que pode ser usado em Prolog para testar se duas variáveis têm valores diferentes. Neste caso, *member* é usado para verificar se um par (Bloco, Posição) pertence a um dado estado. Finalmente *replace* é usado para produzir um estado semelhante ao estado inicial, substituindo a posição de um bloco por outra.

Depois de definir a ação ou ações de transição de estado, é necessário informar o predicado astarps/3 quais são as ações (ou ação), usando o predicado *state_tr_act/3* ou o predicado *state_tr_act/2*:

```
state_tr_act(move(_, _, _, _, _), 1).
```

Este predicado também especifica o custo de efetuar a ação especificada. Neste caso, a ação tem custo 1.

Quando o algoritmo é executado, usa as ações especificadas para produzir os estados descendentes de cada estado considerado, e usa o custo como parte da informação necessária para avaliar cada estado ainda não considerado. No final, devolve uma lista que representa a sequência de ações que permitem ir do estado inicial até ao estado objetivo.

Se pretendermos que as ações contidas nessa lista sejam diferentes das ações usadas pelo algoritmo, por exemplo para reduzir a complexidade, usa-se *state_tr_act/3* em vez de *state_tr_act/2*, por exemplo:

```
state_tr_act(move(_, _, Block, From, To), move(Block, From, To), 1).
```

Neste caso, informamos o algoritmo que, na solução, basta indicar, para cada ação usada, apenas os argumentos correspondentes ao bloco que é transportado, a posição onde ele estava antes de ser transportado, e a posição para onde ele é transportado.

Para usar o *astarps/3*, é preciso também definir e especificar o predicado que testa se um determinado estado é um estado objetivo. O predicado é definido em Prolog, por exemplo:

```
goal(State):-
    member((a, b), State),
    member((b, c), State).
```

Um estado *State* é um objetivo desde que tenha o bloco A sobre o bloco B e o bloco B sobre o bloco C. Agora tem de se informar o algoritmo que o predicado *goal* é o predicado que testa de um estado é um estado objetivo. Para isso, usa-se o predicado *astarps_goal_test_predicate/1*, por exemplo:

```
astarps_goal_test_predicate(goal).
```

Finalmente, é necessário definir e especificar o predicado que estima o custo de chegar de um determinado estado até ao estado objetivo. No exemplo que temos estado a usar, vamos dizer que o custo estimado de um estado é 1, se esse estado não for o objetivo; e é 0, se esse estado for o objetivo:

```
hcost(State, 1):- \+ goal(Sate).
hcost(State, 0):- goal(State).
```

E agora informamos o A* que deve usar o predicado *hcost/2* para fazer a estimativa dos custos, neste caso:

```
astarps_heuristic_predicate(hcost).
```

Depois das definições e especificações necessárias, podemos experimentar o algoritmo, usando o predicado astarps/3:

```
?- astarps([(a, b), (b, 1), (c, 3)], Solution, Cost).
Solution = [move(a, b, 4), move(b, 1, c), move(a, 4, b)]
Cost = 3
```

Na interação acima, pedimos ao A* para nos dizer que é necessário fazer para obter uma pila em que o bloco A está sobre o B, e em que este está sobre o C, partindo de um estado em que o bloco A está sobre o bloco B, o qual está no espaço 1, e em que o bloco C está no espaço 3. O A* diz-nos que basta mover o bloco A do bloco B para o espaço 4, depois mover o bloco B do espaço 1 para cima do bloco C, e depois mover o bloco A, do espaço 4 para o bloco B. Mais ainda, a solução apresentada tem um custo de 3 unidades.

2 Usar o predicado *astar/7*:

astar(Init, Expansion, Heuristic, Cost, GoalTest, Solution, SCost), em que

Init é o estado inicial (ou o nó de partida do grafo). *Init* é dado ao predicado *astar/7*.

Expansion é um predicado que, dado um estado gera uma lista com todos os estados descendentes do estado recebido. *Expansion* é dado ao predicado *astar/7*.

Heuristic é um predicado que recebe um estado e determina uma estimativa do custo para chegar desse estado a um estado objetivo. *Heuristic* é dado ao predicado *astar/7*.

Cost é um predicado que recebe dois nós adjacentes (primeiro o pai, depois o filho), e determina o custo de passar do nó pai para o nó filho. Cost pode também ser definido como um predicado que recebe uma ação (ou um arco) e devolve o seu custo. *Cost* é dado ao predicado *astar/7*.

GoalTest é um predicado que verifica se um determinado estado é um objetivo. *GoalTest* é dado ao *astar/7*.

Solution recebe a solução do problema, isto é, a lista de ações que, se forem aplicadas ao estado inicial, originarão o estado objetivo. *Solution* será produzida pelo algoritmo.

SCost recebe o valor do custo da solução encontrada. *SCost* será produzida pelo algoritmo.

Exemplo de utilização do predicado *astar/7*:

```
test5(Solution, SCost):-
      astar([0, 1, 3, 4, 2, 5, 6, 7, 8], p8successors,
            mdist2goal, gcost, goal_state, Solution, SCost).
```

[0, 1, 3, 4, 2, 5, 6, 7, 8] representa o estado inicial. Neste caso representa as peças num puzzle de 8 peças. O 0 representa o espaço não preenchido. Neste problema, decidimos representar cada estado por uma lista de 9 posições, mas poderíamos ter usado qualquer outra representação porque o algoritmo nunca tem de lidar com a representação dos estados. São os predicados *p8sucessors/2*, *mdist2goal/2*, *gcost/3* (ou *gcost/2*) e *goal_state/1* que processam os estados deste problema específico. Desta forma, o algoritmo mantem-se totalmente independente de qualquer problema.

p8successors é o nome do predicado que recebe um estado de um problema do 8-puzzle e gera uma lista com todos os estados sucessores do estado recebido. Para usar o A* para resolver problemas do 8-puzzle, tem de se implementar o procedimento *p8successors/2*.

mdist2goal é o nome de um predicado que recebe um estado de um problema de 8-puzzle e determina o custo previsto para chegar desse estado ao estado objetivo. O seu nome refere-se à distância de Manhattan (mdist) do estado ao objetivo (2goal). *mdist2goal/2* tem de ser implementado antes de ser passado ao algoritmo.

gcost é o nome de um predicado que recebe dois nós adjacentes, no 8-puzzle, e devolve o custo para transitar de um para o outro. Em problemas em que se define o custo das ações, em vez de *gcost/3*, define-se *gcost/2* que devolve o custo de cada ação. *gcost/3* (ou *gcost/2*) tem de ser implementado para poder ser usado no algoritmo.

goal_state é o nome de um predicado que verifica se o estado recebido é um estado objetivo. *goal_state/1* tem de ser implementado para que possa ser usado no algoritmo.

2.1 Expansão de um estado

A operação de expansão de um estado recebe o estado a expandir e produz a lista dos seus sucessores diretos, os quais resultam da aplicação de todas as operações de transição de estado que se puderem aplicar ao estado recebido. A operação de expansão é, em geral, a operação mais complexa que tem de ser passada ao algoritmo. No caso em que o *astar/7* é usado para resolução de problemas, cada sucessor do estado expandido é uma estrutura de dados constituída por um estado e pela ação que foi aplicada ao estado expandido para originar esse sucessor: *Ação-Estado*[1].

No 8-puzzle, definimos quatro ações de transição de estado: cada uma dessas ações move o espaço em branco (representado pelo algarismo 0) num dada direção: *move_up/2*, *move_down/2*, *move_left/2* and *move_right/2*. Cada uma destas ações recebe um estado e produz o estado que se obtém de mover o espaço não preenchido para a sua nova posição, por exemplo:

```
?- move_up([1, 2, 3, 7, 4, 0, 8, 5, 6], E).
E = [1, 2, 0, 7, 4, 3, 8, 5, 6]
```

Cada uma das ações de transição de estado verifica se as precondições para a sua aplicação se verificam. Se as precondições para a execução da ação não forem verdadeiras, a ação falha. Por

[1] No caso em que o *astar/7* é usado para determinar o caminho entre dois nós, num grafo, o resultado de expandir um nó é a lista dos nós acessíveis ao nó expandido através de um único arco. Neste caso, os sucessores de um nó são apenas nós; não são estruturas *arco-nó*.

exemplo, *move_up/2* só pode ser executado se a posição do 0 no estado recebido não for nem 1, nem 2, nem 3, i.e., se o espaço não estiver na primeira linha.

Para que a definição da operação *p8successors/2* fosse simples, definiu-se uma ação geral de transição de estado: *transop/3*. Basicamente, *transop* define-se como *move_up* ou *move_down* ou *move_left* ou *move_right*. Além disso, *transop/3* tem um argumento extra em relação a cada uma das ações definidas: um termo representando a própria ação que será executada.

```
% transop(State1, State2, Op): state transition operator
% Op is the action that transforms State1 into State2
transop(State1, State2, move_right(State1, State2)):-
     move_right(State1, State2).

transop(State1, State2, move_left(State1, State2)):-
     move_left(State1, State2).

transop(State1, State2, move_up(State1, State2)):-
     move_up(State1, State2).

transop(State1, State2, move_down(State1, State2)):-
     move_down(State1, State2).
```

Tendo uma única operação de transição de estado, é muito fácil definir a operação *p8successors/2*:

```
p8successors(State, Successors):-
     findall(Op-State2, transop(State, State2, Op), Successors).
```

2.2 Previsão do custo de um estado

A implementação da heurística usada para prever o custo mínimo de um estado, isto é, o custo mínimo de chegar do estado em causa ao estado objetivos, poderá ser também razoavelmente complexa, dependendo da heurística escolhida.

No nosso caso, o custo previsto de um estado é a distância de Manhattan entre esse estado e o estado objetivo. Como temos o predicado *goal_state/1* que devolve o estado objetivo, podemos usá-lo na definição da heurística *mdist2goal/2*:

```
mdist2goal(State, MD):-
     goal_state(Goal),
     mdist(State, 0, Goal, MD).
```

mdist/4 devolve a distância de Manhattan entre dois estados (*State* e *Goal*) do 8-puzzle. O segundo argumento, inicializado com zero, é usado para determinar a posição da primeira peça do estado recebido *State*.

mdist/4 percorre a lista que representa o estado *State* e calcula, para cada peça, a sua distância de Manhattan entre a posição que ocupa no estado *State* e a posição que ocupa no estado *Goal*. O predicado devolve a soma das distâncias de Manhattan de todas as peças.

2.3 Os outros predicados necessários

Falta ainda comentar a implementação de dois predicados: *gcost/2* e *goal_state/1*.

Assumimos que todas as ações – *move_up*, *move_down*, *move_left* e *move_right* – têm custo unitário.

```
gcost(_, 1).
```

A implementação de *goal_state/1* é igualmente simples:

```
goal_state([1, 2, 3, 4, 5, 6, 7, 8, 0]).
```