Prolog Exercices

Intelligent Systems

David Montalvo García, Pablo Martínez Pérez

December 5th, 2018

1 Exercise 1 (numbers.pl).

Authors
David Montalvo García
Pablo Martínez Pérez

Version 1.0

Determine if a list contains a number in its elements.

is_number(+*List:list***)**

Checks if the list given has a number.

Examples:

```
?- is_number([a, b, 1, c, d]).
true.

?- is_number([a, b, c, d]).
false.
```

Parameters

List *List* provided to look for numbers inside it.

2 Exercise 2 (contributor.pl).

Authors
David Montalvo García
Pablo Martínez Pérez
Version 1.2

Determine if a person is medium contributor or not. The following cases need to be covered:

- Can not be a foreigner
- If married, gross income of both parterns together can not exceed 70.000
- Its private income can not exceed 40.000

married(+*Person1:String, +Person2:String*)

Checks if two people are married.

Examples:

```
?- married(brad_pitt, angelina_jolie).
true.

?- married(javier_bardem, Y).
penelope_cruz.
```

Parameters

Person1	First person to be checked if married.
Person2	Second person to be checked if married.

medium_contributor(+*Person:String*)

Checks if a person is a medium contributor. To be so, it can not be a foreigner and needs to be native. This double checking is done in case someone has double nationality, in which case it is not a medium contributor.

Parameters

Person	Specific name for a person.

meets_income(+*Person:String*)

Checks if a person meets the income requirements to be a medium contributor

Parameters

Person	Specific name for a person.

3 Exercise 3 (menu.pl).

Authors
David Montalvo García
Pablo Martínez Pérez

Version 1.2

Shows a menu to allow the user to execute the A* Algorithm or Lowest Cost First Algorithm.

search

The user is shown a menu to choose between 2 search algorithms. There are 3 possiblities:

1. Exit: `menu(0)` is called and the program finishes.
2. Lowest Cost First: `menu(1)` is called and the user is asked for more information.
3. A*: `menu(2)` is called, another menu is shown and the user is asked for more information.

menu(+*Num:int*)

Menu that asks the user to introduce input in order to get the start state and the goal state.

Parameters

Num	Index that determines which heuristic will be used to compute the path.

execute_algorithm(+*Option:int*)

This predicate calls `read_states/2`, and adds to the knowledge base the heuristic value. Then executes the `a_star_algorithm/4`, displays the results and removes from knowledge base the stored heuristic value.

Parameters

Option	Index determing which heuristic is added to the knowledge base

read_states(-*Start_state:list, -Goal:list*)

Asks the user to introduce a specific State and a *Goal* state. Both need to be lists. Aftweards it checks both states are valid (both lists need to have the same elements).

Parameters

Start_state	Start State that the user is ask to provide.
Goal	*Goal* state that the user is ask to provide.

cls

Cleans the screen.

4 A* Algorithm (star.pl).

Authors
David Montalvo García
Pablo Martínez Pérez

Version 1.7

This module defines predicates to execute the A* (A-star) search algorithm, and some predicates related to various heuristics functions.

a_star_algorithm(+*State:list, +Goal:list, -Path:list, -Num_explored_nodes:int*)

Returns the optimum path from the *State* given to the *Goal* for the panckes relaxed problem, and the number of nodes explored in *Num_explored_nodes*.

Examples:

```
?- a_star_algorithm([3,1,4,2], [4,3,2,1], Path, Explored_nodes).
Path = [[3, 1, 4, 2], [3, 1, 2, 4], [4, 2, 1, 3],
  [4, 3, 1, 2], [4, 3, 2, 1]],
Explored_nodes = 15.

?- a_star_algorithm([2,5,3,1,4,6], [3,4,6,2,1,5], Path).
Path = [[2, 5, 3, 1, 4, 6], [2, 5, 3, 1, 6, 4], [2, 5, 3, 4, 6, 1],
  [2, 5, 1, 6, 4, 3], [3, 4, 6, 1, 5, 2], [3, 4, 6, 2, 5, 1],
  [3, 4, 6, 2, 1, 5]],
Explored_nodes = 239.
```

Parameters

State	Current state.
Goal	*Goal* state.
Path	*Path* found (which is optimum due to A*).
Num_explored_nodes	Number of nodes explored.

search(+*Goal:list, +Frontier:list, +Explored_states:list, -Path:list, -New_explored_states:int*)

Returns the solution path and the number of nodes explored. The first node of the frontier is extracted and compared to the *Goal* node. The following cases are possible:

- If the extracted node matches the target node, the path associated with the node is returned.
- Otherwise, the current node is expanded and added to the border using the predicate `add_to_frontier/5`. The current node is added to the list of explored nodes and `search/4` is called again with the new ordered border and the new list of expanded nodes.

If the frontier is empty, the solution is not found and the empty path is returned.

The nodes are represented by lists of three elements:

```
Node = [Node_state, Node_path, G_value]
```

where Node_path is the current path to the node defined as followed:

```
Node_path = [[State1], [State2], [State3], ..., [StateN]]
```

Parameters

Goal	List of the goal state.
Frontier	List of frontier nodes.
Explored_states	List of explored states.
Path	List of founded path.
Num_explored_nodes	Number of nodes explored.

add_to_frontier(*+Frontier:list, +Node:list, +Goal:list, +Explored_states:list, -New_ordered_frontier:list*)

Adds the child nodes of *Node* to *Frontier* if they have not been previously explored, and returns this new frontier in the variable *New_ordered_frontier*. To do so, the following steps are carried out:

1. The list of child nodes of *Node* is obtained using `expand/3`.
2. New nodes are added to the border.
3. A list (called Frontier_values_inversed) is generated with the values of the evaluation function applied to each node of the frontier, but inverted. This inverted list is inverted again to obtain the f-values in the right order, and stored in Frontier_values.
4. Using the predicate `pairs_keys_values/3` (available in library pairs) each node of the border is associated to its f-function value. This set of key-value pairs is stored in the variable Pairs_key_value.
5. Finally, the list of keys-value Pairs_key_value is sorted using `keysort/2` predicate, and the f-function value is removed from the list using `pairs_values/2`. The list of already ordered nodes is stored in *New_ordered_frontier*.

Parameters

Frontier	List of nodes to be explored, sorted by evaluation function.
Node	Current node which is being evaluated and might be added to *New_ordered_frontier* if it has not already been explored.
Goal	*Goal* state.
Explored_states	List where all explored states are stored.
New_ordered_frontier	*Frontier* which contains the new child nodes in order (depending on `g()`).

expand(+*Node:list*, +*Explored_states:list*, -*Childs:list*)

Returns a list with the child nodes of *Node* that have not yet been explored in the variable *Childs*. To do this, `expand/4` is used.

Parameters

Node	Current node which is being expanded.
Explored_states	List where all explored states are stored.
Childs	Child nodes of *Node* that have not yet been explored.

expand(+*Node_state_tail:list*, +*Node:list*, +*Explored_states:list*, -*Childs:list*)

Creates a list of nodes with the child nodes of *Node* that have not yet been explored, using for this purpose the *Explored_states* variable. Recursion is used to cover all possible child states of Node_state, by decreasing the size of the *Node_state_tail* with each recursive call.

For each recursive step, the child state is generated by the predicade `generate_child/3`, and it is checked if the generated state is alredy in the list *Explored_states*, using `member/2`. Two cases may occur:

1. The node has already been explored. In this case the node is not generated.
2. The node hasn't been explored yet. In this case, the cost of switching from the current state to the child state is calculated using `g_cost/3`, and this cost is added to the accumulated cost of *Node*. Finally:
   - The child state is added to the path of the parent node.
   - The child node is created as a list composed of child state, the previously calculated path and the accumulated cost of the node.
   - This new node is added to the returned *Childs* list.

Example:

```
?- expand([3,1,2,4], [[3,1,2,4],[[3,1,4,2],[3,1,2,4]],2],
          [[3,1,4,2],[3,1,2,4]], Childs).
Childs = [[[3, 4, 2, 1], [[3, 1, 4, 2], [3, 1, 2, 4], [3, 4, 2, 1]], 5],
          [[4, 2, 1, 3], [[3, 1, 4, 2], [3, 1, 2, 4], [4, 2, 1, 3]], 6]].
```

Parameters

Node_state_tail	Tail of the given state which decreases with each recursive call.
Node	Current node which is being expanded.
Explored_states	List where all explored states are stored.
Childs	Child nodes of *Node* that have not yet been explored.

generate_child(+*State_tail:list*, +*State:list*, -*Child_state*)

Creates a child of a *State* using a tail from it. It substracts the tail from *State* and afterwards adds the reversed tail to *State*.

Examples:

```
?- generate_child([1,2,3,4], [1,2,3,4], Child).
Child = [4 ,3, 2, 1].

?- generate_child([3,4,5], [1,2,3,4], Child).
false.
```

Parameters

State_tail	Tail of the given state used to generate the child.
State	The state from whom the child is generated.
Child_state	Child state generated.

g_cost(+*State1:list*, +*State2:list*, -*Cost:list*)

Determines the cost of moving from one state to another for the relaxed pancake problem (how many pancakes are raised to make a movement).

Examples:

```
?- g_cost([1,2,3,4,5], [1,2,5,4,3], Cost).
Cost = 3.

?- g_cost([1,2,3,4,5], [1,2,3,4,5], Cost).
Cost = 0.

?- g_cost([1,2,3,4,5], [5,2,3,4,1], Cost).
Cost = 5.
```

Parameters

State1	First list given with the pancakes position.
State2	Second list given with the pancakes position.
Cost	*Cost* of moving from *State1* to *State2*.

matching_point(+*List1:list*, +*List2:list*, -*Index:list*)

Determines up to which index two lists exactly match.

Examples:

```
?- matching_point([1,2,3,4], [1,2,4,3], Index).
Index = 0+1+1.

?- matching_point([1,2,3,4], [1,2,3,4], Index).
Index = 0+1+1+1+1.

?- matching_point([4,2,3,1], [4,1,3,2], Index).
Index = 0+1.
```

Parameters

List1	First list given that is to be compared.
List2	First list given that is to be compared.
Index	*Index* determing until which position *List1* and *List2* exactly match.

evaluate_nodes(+*Frontier:list, +Goal:list, -F_values_inversed:list***)**

For each Node in the frontier, it calculates its heurstic value using `heuristic_value/3` recursively. It then returns a list of integers containg the f (g + h) value for each node. This list is in reversed order.

Parameters

Frontier	The frontier containing the nodes for which the f value are to be evaluated.
Goal	*Goal* node.
F_values_inversed	Reversed list containing the f values for each node in *Frontier*.

heuristic_value(+*State:list, +Goal:list, -H_value:int***)**

Computes the heursitic value given for two states given. There are three possible heurisitcs:

1. 0 regardless of the two states provided if `heurisitc(0)` is true.
2. h1 if `heuristic(1)` is true.
3. h2 if `heuristic(2)` is true.

Parameters

State	*State* provided.
Goal	*Goal* node.
H_value	Heuristic value computed for the two states given.

add_heuristic_option(+*Option:int***)**

Add to the knowledge base the following fact: `heuristic(Option).`

Parameters

Option	Number of the heuristic to be added

delete_heuristic_option(+*Option:int***)**

Delete to the knowledge base the following fact: `heuristic(Option).`

Parameters

Option	Number of the heuristic to be deleted

h2(+*State:list,* +*Goal:list,* -*Num:int*)

Calculates the heuristic value of the current state as the sum of the distances between the current position of each pancake and its current position, minus the number of misplaced pancakes.

Parameters

State	Current state.
Goal	*Goal* state.
Num	The heuristic value, which corresponds to the number of pancakes poorly placed.

h1(+*State:list,* +*Goal:list,* -*Num:int*)

Calculates the heuristic value of the current state as the number of misplaced pancakes.

Parameters

State	Current state.
Goal	*Goal* state.
Num	The heuristic value, which corresponds to the number of pancakes poorly placed.

gap_to_goal(+*State:list,* +*Goal:list,* -*Num:int*)

For a given state it determines the heuristic cost of moving to *Goal*. To do so, it recursively determines the cost for each Element in *State* and sums them all using `min_gap/4`.

Parameters

State	Current state.
Goal	*Goal* state.
Num	Heuristic value of moving from *State* to *Goal*.

min_gap(+*Element:Generic,* +*State:list,* +*Goal:list,* -*Min:int*)

Calculates the minimun gap between the position of an element in the *State* list and its position in *Goal* list. These steps are followed:

1. The index in which *Element* is in the *State* is saved in State_index.
2. The index in which *Element* is in the *Goal* is saved in Goal_index.
3. The absolute value of the difference between State_index and Goal_index is calculated. This value is saved in Min1.
4. The lenght of *Goal* is saved in List_length (Which is the same for *State* and *Goal*).
5. The difference between List_length and Min1 is calculated. This value is denoted by Min2. Min2 denotes the cyclic distance.
6. Take as a *Min* the minimum between Min1 and Min2. The minimun value between distance and the ciclical distance is returned.

Parameters

Element	An element (should be in both state and *Goal*).
State	Current state.
Goal	*Goal* state.
Min	The minimun gap between the position of an element in the *State* list and its position in *Goal* list. That is, the number of movements needed to move an *Element* from *State* to *Goal* allowing cyclical movements.

position(*+State:list, +Element:list, -Index:int*)

Returns the position (index) of the first appearance of an element in a list or false if the element is not found. It differs from nth0 as it only returns the first appearance.

Parameters

State	List in which the element will be searched for.
Element	*Element* to be search for.
Index	Position of the *State* in the list (*Index* of element in list)

5 See online documentation (documentation.pl).

Authors
David Montalvo García
Pablo Martínez Pérez

Version 1.0

This prolog script starts documentation server at port 4000 and opens the user's default browser on the running documentation server. To see the code, click on the orange icon on the right side of each predicate.