Practical "Introduction to Artificial Intelligence"

Prof. Dr. Gunter Grieser

Block 1: Prolog

Sheet 6: A* algorithm

Hints:

- *In Block 1 (Prolog) you do not have to submit your solutions to me. Just solve the excercies and discuss your problems and solutions.The aim of Block 1 is that you become familiar with the prolog programming.*
- *If you do not succed with a task, just delay it and try it again later. Some constructs need time to settle in the brain and will become easier as you get more experienced.*

Excercise 6.1

Visit the site http://qiao.github.io/PathFinding.js/visual/

a) Get familiar with the GUI and experiment with the different algorithms

- breadt first search
- best first search
- A*
- IDA*

b) For each algorithm, build structures so that the algorithm find solutions

- as fast as possible
- as slow as possible

Excercise 6.2

We now incrementally implement the A* algorithm for the problem "Towers of Hanoi":

- you have n disks of different size (initially: start with n=3) and three rods (named A, B, and C)
- Initially, all disks are on the leftmost rod (A), starting by the largest up to the smallest.
- The target is to move all disks to the rightmost rod (C), where the following rules apply:
  - Only one disk can be moved at a time.
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
  - No disk may be placed on top of a smaller disk.

a) Define the search problem (see lecture slide 11 of deck 3), i.e. how do you model the problem?

b) Invent at least one heuristics (better: at least two) for the search problem. Which of your heuristics is admissable?

c) We model a certain <u>state</u> in prolog as follows:

- A disk is described by a positive integer. I.e. `5` means the disk of diameter 5.
- A state is modeled by the term `state(A,B,C)`, where `A, B,` and `C` are lists of disks, where the upper disk is in front.
  - Example: `state([1,2,3], [], [])` denotes the initial state, where 1 is on 2 is on 3 on rod A.

Write the following predicates:

- a predicate `valid_state(State)` that is true if `State` is valid, i.e. there is no rod where a larger disk is on top of a smaller one.
- a predicate `final_state(State)` that is true if `State` is a final state.

d) We model a certain <u>node</u> of the search tree in prolog as follows:

- `node(State, F, G, H, Path)`, where
  - `State` is a state (see c),
  - `F, G, H` are integers (the values of *f(n)*, *g(n),* and *h(n)*)
  - `Path` is a list of moves that lead to that node
    - example: `[ab, ac, bc]` means that first, we took the upper disk from rod A to rod B, then the upper disk from A to C and then the disk from B to C.
  - Example: `node(state([1,2,3], [], []), 0, 0, 0, [])` denotes the initial node for the search, i.e. inital state, $f(n) = 0 + 0$ and empty path.

Write the following predicates:

- a predicate `final_node(Node)` that is true if `Node` is a final node.
- a predicate `move_A2B(NodeBefore, NodeAfter)` that is true if `NodeAfter` is a node resulting from moving the top disk from rod A to rod B.
  - if it is not possible to move from A to B (because A is empty or we would put a larger disk on a smaller one), then this predicate fails.
    - *Hint: you may want to use your predicate `valid_state/1` from c).*
  - *g(*`NodeAfter`*)* = *g(*`NodeBefore`*)* + *1* (i.e. each move has cost 1.)
  - *h(*`NodeAfter`*)* = *0* (I.e. we don't use a heuristics in the moment.)
  - *f(*`NodeAfter`*)* = *g(*`NodeAfter`*)* + *h(*`NodeAfter`*)*
  - `PathAfter` is `Pathbefore` extended by `ab`.
- Analogous, write predicates `move_A2C, move_B2A, move_B2C, move_C2A,` and `move_C2B`
  - *For advanced: Try to write the predicates in an abstract manner, instead of programming "Move from A to B" and so on, model it as "Move from X to Y" and then instantiate it.*
- a predicate `successor_nodes(Node, ListOfSuccessors)` that computes the list of all successors.

  - *Hint: Call `move_A2B(Node)` and if it succeeds, add it to the result list, then call `move_A2C` and so on. For advanced: you can use `bagof/3`.*

e) We now program the search algorithm.

- The fringe is modeled as sorted list of nodes, the first element is the one with the least *f*-value.

Write a predicate `search(Fringe, Path)` that implements the search loop as follows:

- Check whether the first element of the `Fringe` list is a final node. In this case, sucess with the `Path` from the first element..
- Otherwise:
  1. Remove the first node *n* from `Fringe`.
  2. Compute all successors from *n*. (using `successor_nodes` from d)
  3. Insert each successor into the `Fringe` so that the list is sorted by ascending *f*-values.
     - *Hint: you can implement it in a "bubble sort" manner, i.e. inserting each successor node separately. Alternatively for the advanced: use `sort/4`.*
  4. For tracing: print out the currrent fringe (use `write/1`, `nl/0` or, for advanced: `format/2`)

Which algorithm from our lecture is implemented by this?

f) Now make some experiments with your implementation.

- Insert the following predicates into your program (these call your `search/2` with a start node of certain size, count nodes, and print some statistics):

```
search(Problem_Size):-
        % --- Initialization
        set_prolog_flag(numberOfCreatedNodes,0),                    % initiate counter
        findall(Num, between(1, Problem_Size, Num), List_1_to_N),   % create a list 1... Problem_Size
        Initial_Node = node(state(List_1_to_N, [], []),0,0,0,[]),   % create initial node
        statistics(cputime,Start_Time),
        % --- Do the search
        search([Initial_Node], Path),!,
        % --- Print statistics
        statistics(cputime,End_Time),
        Time = End_Time - Start_Time,
        current_prolog_flag(numberOfCreatedNodes, N),
        length(Path,L),
        format('Search finished. Created ~d nodes in ~1f CPU-seconds. Solution has ~d moves.~n ',
                [N, Time, L]).

increase_node_counter(Nodes):-
        length(Nodes,L),
        current_prolog_flag(numberOfCreatedNodes, N),
        N1 is N + L,
        set_prolog_flag(numberOfCreatedNodes, N1).
```

- Expand your program for `search/2` by a line that counts the number of created nodes: `increase_node_counter(SuccessorNodes),`
  - where `SuccessorNodes` is the list of all successors created in step 2.
- Now run `search(3)`, `search(4)`, ... and observe the statistics.

g) Implement your heuristics from b) and build it into your `search/2`.

- Repeat the experiments from f). Do you get other results? Which? Why?