UNIVERSITEIT GENT

Ghent University
Faculty of Engineering and Architecture
Knowledge Based Systems and Artificial Intelligence

Report Project Expert Systems: Route Planning in CLIPS

Tom Naessens
Dieter Decaestecker

2nd master Computer Science Engineering

Friday October 31, 2014

1 Algorithm description

1.1 General description

All neighbourhood space search algorithms can be summarized as follows: one starts with a point, designated by some coordinates, and iteratively searches the space around that point for a goal, according to some pattern or rule. This pattern or rule defines how the iterative search selects following points and expands the search space. As indicated in the assignment, we use the A-star algorithm to search select candidate search paths.

The search space exists here of a 2D grid, where nodes are possibly connected to a horizontal and/or vertical neighbour. At the start of the execution, the user selects a start node and a goal node. We then search the corresponding coordinates (x and y positions) of both nodes, to be able to position them on the search plane. The next step is to build the search tree. We start setting up this search tree by adding the start node as the root node. We then continue to complete our search tree by adding the neighbours of the root (start) node as children, and select the best node to continue our search following the A* heuristic (more formally described in subsection 1.2). Nodes of which the children have been expanded and added to the search tree are marked as closed.

This process repeats recursively, until the goal node has been expanded as a child of a node in the tree, indicating that we have found our solution. The path from the root node to the goal node can then be traced back by iterating the search tree bottom op, starting with the goal node, selecting the parents of each node in the process. When all nodes have been marked as closed and the goal node has not been added to the search tree, there exists no path between the start and the goal node.

1.2 Heuristic implementation

As described in the assignment, the heuristic used to determine which open node should be selected in the search tree, is the A* heuristic. This heuristic tries to minimize the cost of the path so far, and the predicted cost to the goal from that node. This results in the following formula

$$f(n) = g(n) + h(n) \tag{1}$$

where $f(n)$ is the estimated total cost, $g(n)$ the cost of the path so far and $h(n)$ the estimated cost to the goal.

$g(n)$ can easily be filled in, as this is the depth of the node in the search tree. $h(n)$ however depends on the dimension of the search space and the connections between different nodes. As we are working in a grid-like structure, where a node can only be connected to its horizontal en vertical neighbours, the shortest distance between two points is the Manhattan distance. This information changes formula 1 to

$$f(n) = \text{depth}(n) + (|n_x - g_x| + |n_y - g_y|)$$

where $depth(n)$ is the depth of the node n in the search tree, and g is the goal node, with corresponding x and y-coordinates.

2 Code explanation

2.1 Remarks

In the attached file, `decaesteckerd-naessenst-path.clp`, our complete ruleset can be found with inline comments. The purpose of this section is to elaborate on changes we have made to the existing rules, and to describe the new rules we added, including why we added these rules.

We have also added a `printout` statement to each rules' right hand side, as this provides a better overview on what happens internally and which rules are invoked at what point in time. These added `printout` rules are prefixed with "`DEBUG:`".

2.2 Adapted rules

To be able to solve the assignment efficiently, some rules and definitions from the given `path.clp` file had to be adapted. The changes made to this file are the following:

- **Addition of $f(n)$ `field` to the `node` `template`:** As nodes are stored in the search tree, and generally do not switch positions later, it is more efficient to calculate the $f(n)$ `value` of that node once, without calculating it every time a new `node` is selected as active.

- **Addition of LHS condition in the `generate-neighbours` rule:** Nodes which are already present in the search tree should not be added again, as they (in this 2D grid situation, not in all situations!) will always have a higher $g(n)$ value and have the same $h(n)$ value, leading to a higher $f(n)$. If we would include these duplicate nodes, the duplicate node that was added first with the minimum $f(n)$ would always be selected as the new active node, with as result that the later duplicates would never get chosen.

- **Increase of `salience` in `compose-route` and `print-route`:** Once a goal has been reached and a route has been `assert`ed, we want to immediately compose the route and print it without doing anything else. In the given implementation, it is possible that other rules with a higher `salience` could fire, rendering these executions unnecessary. Increasing the `salience` of both rules, and halting at the end of the `print-route` rule prohibits these unnecessary executions. It is also possible to prohibit this by placing `(not (route))` in the LHS of all rules that could possible be triggered, but this would lead to cluttered and repetitive code.

- **Addition of `NIL` to the `route` LHS in `print-route`:** For some reason, the `compose-route` rule ads `NIL` as the most top-level parent, causing the `print-route` never to fire. Altering the `(route ?d $?q ?b)` condition to `(route NIL ?d $?q ?b)` solves this problem.

2.3 Added rules

The rules given in the `path.clp` file were not enough to fix the problem, and had to be completed. Following is a description of all the added rules. These can also be found with documentation at the bottom of the attached file.

- **Creation of the root node (`Set-start-node`):** When the program is `init`ed, a `start` fact is created. This fact creation should also trigger the creation of the root `node` in the search tree. So, whenever there is a `start` fact and there exists no `node` with the same name as this `start` fact, a new node should be created without parents at depth 0 of the search tree.

- **Addition of a stop (failure) rule (`Halt-when-closed-empty`):** When all the nodes are closed and the goal hasn't been reached (so there is no found route), the search space has been depleted. The execution should then stop and print "`FAILURE`" to the screen.

- **Calculation of the $f(n)$ values of new nodes (`Update-fn` and `calculate-fn`):** As mentioned in subsection 2.2, we save the $f(n)$ value of nodes in the search tree in the `node template`. Every time a new node is added to the search tree, it's $f(n)$ value should automatically be generated. To avoid clutter in our code, we make use of a helper function, `calculate-fn`, the implementation of which can be found in the attached source code.

- **Selection of a new active node (`Select-active-node`):** This probably is the most important rule of the program. Of all open `node`s, the one with the lowest $f(n)$ value should get selected. To manage this, we make use of the following condition in the LHS of this rule: `(not (node (fn ?fn&:(< ?fn ?open-fn)) (status open)))`. Translated, this means that there should not be another open `node` of which the $f(n)$ is less than the selected `node`. We put this found `node` to active, to enable the further expansion of the search tree by the `generate-neighbours` rule.

- **Generate a `route` fact when the goal has been reached (`Reach-goal`):** When a `node` has been generated with the same name as the `goal` fact, a `route` fact should be generated, allowing the `Compose-route` and `Print-route` rules to take over and halt the execution when the route has been traced back.

3 Sample execution

To execute the code, one can open the attached file in `CLIPS` and load the buffer into a `dialog` window. After running `(reset)`, all initial facts get generated. When executing `(run)`, the user has the ability to pick the name of the `start` and `goal` fact. For example, when searching a path from `I` to `H`, the following output, mentioned in listing 1 is generated.

Listing 1: Sample output from `I` to `H`

```
CLIPS> (reset)
CLIPS> (run)
Init
The grid name of the start position? I
The grid name of the end position? H
*** SNIP: DEBUG OUTPUT ***
SUCCESS: Reached H
*** SNIP: DEBUG OUTPUT ***
The optimal path from I to H passes (C A B D F G)
CLIPS>
```