Knowledge based systems and artificial intelligence

Project Expert Systems: Route planning in CLIPS

Path planning is concerned with the problem of moving an entity from an initial configuration to a goal configuration. The resulting route may include intermediate tasks and assignments that must be completed before the entity reaches the goal configuration. Path planning algorithms can be classified as either global or local. Global path planning takes in to account all the information in the environment when finding a route from the initial position to the final goal configuration. Local planning algorithms are designed to avoid obstacles within a close vicinity of the entity; therefore, only information about nearby obstacles is used. This project will refer to global path planning algorithms where the entire path is generated from start to finish before the entity makes its first move.

Path planning includes both point-to-point and region filling operations. Point-to-point path planning looks for the best route to move an entity from point A to point B while avoiding known obstacles in its path, not leaving the map boundaries, and not violating the entity's mobility constraints. This type of path planning is used in a large number of robotics and gaming applications such as finding routes for autonomous robots, planning the manipulator's movement of a stationary robot, or for moving entities to different locations in a map to accomplish certain goals in a gaming application. While point-to-point operations are appropriate for some applications, they do not always produce the desired route for tasks such as vacuuming a room, plowing a field, or mowing a lawn. These types of tasks require region filling path planning operations that are defined as follows:

1. The mobile robot must move through an entire area, i.e., the overall travel must cover a whole region.
2. The mobile robot must fill the region without overlapping paths.
3. Continuous and sequential operation without any repetition of paths is required of the robot.
4. The robot must avoid all obstacles in a region.
5. Simple motion trajectories (e.g., straight lines or circles) should be used for simplicity in control.
6. An "optimal" path is desired under the available conditions.

Both types of planning operations require searching through all the possible routes to find the most optimal, or efficient route among all the possibilities. For instance, in a robotics application that must move a robot between five locations in the shortest amount of time, the path planner must perform a search to determine in what order to complete these tasks. Conventional AI search algorithms in conjunction with the fields of Graph Theory are used to accomplish this task. The idea of a graph is that nodes are placed at choice points and edges connect each node in the graph. A choice point is anywhere in the graph where a decision must be made as to where to go next. In the example above, choice points would be placed at the initial location of the robot and the five locations that it must visit. The search begins at the choice point placed at the initial location, or node, and ends when all choice points have been reached. The following search algorithms work from the simplest to the more robust algorithms for searching a graph to find efficient routes.

Breadth-first search. This strategy begins with the start node and then expands every one of its neighboring nodes, and then expands successors. Because all the nodes at depth d are expanded before the nodes at depth d+1, this strategy will always find a solution if it exists. If multiple solutions exist, the shallowest goal state will be found first. One obvious problem to this approach is the memory requirements for large search spaces. This strategy fans out in all directions equally rather than directing its search towards the goal.

Depth-first search. This search algorithm is the complement to breadth-first search; rather than visiting all a node's siblings before any children, it visits all of a node's descendants before any of its siblings. When the search hits a dead end, it goes back and expands nodes at shallower levels. Each successor will be even deeper. This strategy reduces the memory requirements needed by a breadth-first search because only a single path from the root to a leaf node is stored. The problem with depth-first search is that it can get stuck going down the wrong path and either never find a solution or find a solution that is suboptimal.

Best-first search. This strategy uses a heuristic function to direct the search in the direction of the goal. The heuristic function estimates the distance remaining to the goal and expands the nodes with the least remaining cost to the goal. It is the fastest of the forward-planning searches discussed so far, heading in the most direct manner to the goal.

A* search. The best-established algorithm for the general searching of optimal paths is A* (pronounced "A-star"). It uses a heuristic search that estimates the cost to the goal, *h(n)*, and minimizes the cost of the path so far, *g(n)*. Since *g(n)* gives the path cost from the start node to node *n*, and *h(n)* is the estimated cost of the cheapest path from *n* to the goal, we have

f(n) = estimated cost of the cheapest solution through n
Or
$f(n) = g(n) + h(n)$

where *f(n)* is the score assigned to node *n*. Along any path from the start node, *f* will always be non-decreasing provided that the heuristic function never overestimates the cost to reach the goal. Using this knowledge it can be proven that the first solution found must be the optimal one, because all subsequent nodes will have a higher *f*-cost. Furthermore, because it makes the most efficient use of the heuristic function, no search that uses the same heuristic function, *h(n)*, and finds optimal paths will expand fewer nodes than A*.

For 2D path-planning, an easy heuristic for *h(n)* is the straight-line distance between *n* and the goal. This distance is always an underestimation of the real path between *n* and the goal.

Project Assignment

Design rules that find the optimal route between a given start and end point using the A-star algorithm. The route network is given as follows, where the grid positions are given in (X,Y) coordinates :

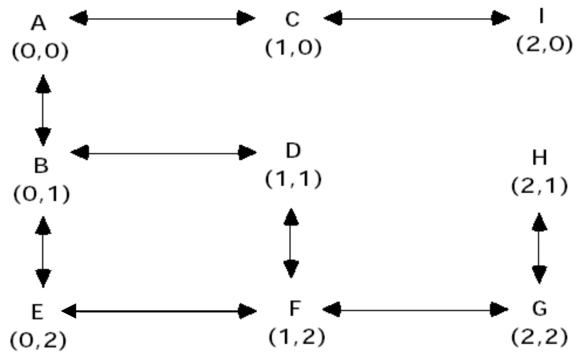

Fig.1. Route network used within the project

This route network is given as an example. The designed rules should be able to find the optimal route in an arbitrary network.

The pseudo-code for the A-star algorithm is given in Appendix A. Note that this code is written for traditional programming languages. You will need to carefully design the coding for a rule-based system.

The file "**path.clp**" contains example data structures and functions which can be used as a starting point. The final program should be able to read the desired start and end point and the shortest path between these two points should be given back as output. The route network in fig. 1 is given as an example. Off course, the designed rules should also give the correct output on other (larger) networks.

The project is done in groups of maximum two students. A report (2-3 pag) should be written which explains the rationale of the designed rules.

The full source code should be commented but is not part of the written report.

The report and the source code should be submitted as a zip-file in Minerva. The file should be named after both authors.
E.g. Rik Bellens & Werner Goeman => bellensr-goemanw.zip

Deadline for submission is **Friday October 31, 2014**

Appendix A

Pseudocode A-STAR

```
Symbols:
        G : search graph
        goal : goal node
        start : start node
        M : list of successors of a search node
        OPEN : list of “open” nodes which need to be explored
        CLOSED : list of “closed” nodes which have been explored
        n,e : search nodes

1) G, OPEN and CLOSED  are initialised as empty
2) Put start in OPEN and as root of G

3) Loop until OPEN is empty. In this case exit with FAILURE (no solution)

4) Extract first n from OPEN. OPEN <- OPEN \ {n}
                              CLOSED <- CLOSED + {n}

5) If (n==goal) retrace path in G and exit with SUCCESS

6) Create M by expanding n to its successors

7) For each e in M
        If e is in CLOSED, M <- M \ {e}
        else
          calculate f(e)
          add e to OPEN ordered according to f
          add e to G as child of n
8) go to 3
```