COM316: Artificial Intelligence
Problem 7: Production Systems

Derin Gezgin — Russell Kosovsky — Jay Nash

The Problem

Intro

We have a fairly good representation that can express numerous facts but our reasoning and action control of these facts has not been formally addressed. There may be several rules with their left-handed side satisfied by facts in the system after any one pass. Some means of determining which will be applicable at what time must be considered. The robot cannot go to both its left and right neighbor at the same time even though both are possible. This is referred to as conflict resolution. It is highly probable, that you have addressed these issues in previous problems as they could not be easily avoided. Other issues may have been part of your design that made it a form of production system. In this problem, we will look deeper into what tools are usually used in the production systems and use them to improve our system.

Given

Same as problem 5, except environment not known in advance.

Problem

Create a production system that can perform our search. Write production rules (like the ones we did for the blocks). We need to have this work for an actual robot. The start position and goal are know, but all else is sensed by the robot. You can assume that there is a function that produces adjacent literals (predicates with variables or points) for all the nodes adjacent to the robot. In addition literals will be added which specify if each of the adjacent nodes are obstacles or not. Do not worry about the high/low stable/unstable issue at this point. Your primary task is to write rules that control the movement of the robot. You need a current, but there should only be one at a time. You need to deal with backtracking when there is nowhere to go. If possible, you should delete previous adjacent literals since they will clutter the facts list and not be needed. You will need to introduce new literals that are there just for control. In my solution, there are 6 rules. In addition to "add" and "delete", I have "execute" as possible consequences. Execute calls a function that executes what is specified by the literal. In my case this happens with the literals `(move_to x)` and `(backtrack_to x)`.

We met to discuss this problem on October 14th

Our Solution

To solve this problem, we should consider multiple situations and ensure that we cover every possible case in the robot's movements:

- Moving to an unvisited, non-obstacle, and adjacent block
- Use a set of heuristics to decide the next node if we would otherwise choose randomly between the adjacent blocks (AKA... we dont know which move is better).
- Backtracking when we don't have any more unvisited squares to go
- Going to the goal if we're adjacent to the goal
- Stopping when the goal is reached
- Stopping if the goal is unreachable

Move to an Unvisited, Non-Obstacle, and Adjacent Block

This is the robot's most basic movement during the search. Considering our rule space, when choosing the next move, we must check if the next location (Y) is adjacent, not an obstacle, and has not already been visited. If we satisfy these conditions, we can update the current to Y, mark Y as visited, and also push Y into our visited_path stack. We would need this stack in the backtracking. A more detailed explanation of its usage is given in the next rule. At the same time, we append adjacents of Y into the fact list *prioritizing lower heuristic values* so that while looping through the fact list, we would first check for better moves. Lastly, the program removes the (adjacent X Y) fact, which is helpful for our last rule, which checks for any possible paths left. The pseudo-code of this rule would be:

Rule 1: Move to an Unvisited, Non-Obstacle, and Adjacent Block

```
if (current X) and (adjacent X Y) and (not (obstacle Y)) and (not (visited Y)) then
        Add: (current Y) (visited Y) (sort (get-adjacent Y));
        Delete: (current X) (adjacent X Y);
        Execute: (move_to Y) (visited_path.push(Y));
end
```

Randomness Reduction Heuristic System

Use a set of heuristics to decide the next node if we would otherwise choose randomly between the adjacent blocks (AKA... we don't know which move is better). If the first heuristic still results in a random choice, use a secondary heuristic, using different heuristics until the choice is not random.

Rule 2: Random Reduction

```
while (node-val-X == node-val-Y) do
    Change heuristic to next heuristic;
    Re-evaluate node values;
end
Execute: (move_to (visited_path.push(better node));
```

Backtrack if Stuck

While going through the grid, our robot might be stuck in a dead end. We have to have a function that catches this case. The function first checks if any adjacents of X have not visited yet. If none exist, the function will return to the last visited location by updating the current. This is where the visited_path stack comes into play as it stores the nodes by order of being visited like a *Depth-First Search*. This rule will fire until the robot has an alternative unvisited adjacent. The pseudo-code of this rule would be:

Rule 3: Backtrack if Stuck

```
if (current X) and (visited all(get-adjacent X)) then
        Add: (current (visited_path.top());
        Delete: (current X);
        Execute: (move_to (visited_path.pop());
end
```

Go to the Goal if it is Adjacent

This rule is fairly direct compared to other rules. In this case, we'd check if the robot is directly next to the goal. If so, it'd move to the goal by updating the current rather than risking having a longer path. Technically, as we implemented a heuristic, this won't be the case but this rule can be considered as a guarantee. The pseudo-code of this rule would be:

Rule 4: Go to the Goal if it is Adjacent

```
if (current X) and (adjacent X goal) then
        Add: (current goal);
        Delete: (current X);
        Execute: (move_to goal) (visited_path.push(goal));
end
```

Stop When the Goal is Reached

This last rule is needed to ensure that our program stops when we reach to a goal. It will basically check if we are on the goal. If so, it'll stop the program and return the complete and clear path which is the visited_path stack. The pseudo-code of this rule would be:

Rule 5: Stop When the Goal is Reached

```
if (current X) and (goal X) then
    Execute: (stop);
    Return: (visited_path);
end
```

Stop If the Goal is Unreachable

There's a possibility that there are no possible paths to the goal. In every iteration of the rules, this rule checks if there are still adjacent facts. If there are not any adjacent fact in the facts list, the program stops executing directly. The pseudo-code of this rule would be:

Rule 6: Stop If the Goal is Unreachable

```
if (find-adjacent-fact) == Null then
    Execute: (stop);
    Return: '( );
end
```