Artificial Intelligence CA

DT282
BSc in Computer Science

Philip Toolan
C17433026

School of Computer Science
TU Dublin – City Campus

04/05/2021

Design Considerations

For the maze I decided to represent it as simple as possible. This meant that making changes to the maze would be simple for the user to do. The number of rows and columns are stored in a variable. Then the start position of the maze is recorded and the final position or goal. The final part of the maze is recording all the positions in the maze that are occupied by a wall. The positions are in the maze are treated as **pos(row, column)**. And the final output of the algorithm is a list of steps needed to be taken to solve the maze, e.g. Up, up, left, down.

To run the algorithm, set up prolog to consult the desired algorithm then type start.
To change the maze, you can edit the beginning of the files to have different start and final positions, obstacles and overall size. Ensuring to put row first followed by column.

Common Predicates

Each algorithm has several predicates that are the same across all 3. These are allowed(), move(), maxDepth() and cost().
Allowed lets the algorithm check if a move in a certain direction is possible, it will not be possible if the position is outside the maze or is occupied by a wall. Move allows the algorithm to move to the new position.
Iterative Deepening specific: MaxDepth holds the maximum depth allowed in the current domain.
Astar specific: heuristic() is used to calculate the heuristic value. Manhattan distance is used in this case. A_star_comparator is used to sort the costs of moving to new nodes for the astar algorithm. Cost is used as a function that calculates the cost of moving from the current position to the new position.

Depth-First Search

The algorithm gets the first node and checks what actions can be performed with this node. It then moves (action) in the direction it can, a new node is taken in. This node is checked against the list of nodes that have already been visited, if it has not then the algorithm will continue to the next node. The node is added to the list of visited nodes.

```
start:-
    depthFirst(S).


depthFirst(Sol) :-
  startPosition(S),
  depthSearch(S, Sol, [S]),
  write(Sol).


depthSearch(S, _, _) :- finalPosition(S), !.
depthSearch(S, [Action|OtherActions], VisitedNodes) :-
  allowed(Action, S),
  move(Action, S, NewS),
  not(member(NewS, VisitedNodes)),
  depthSearch(NewS, OtherActions, [NewS|VisitedNodes]).
```

Solutions:

```
% c:/Users/Legion67/Documents/Prolog/depth_first_p.pl compiled 0.00 sec, 22 clauses
?- start.
[down,down,left,left,up,up,up,up|_8104]
true .

?-
```

Figure 1. Maze 1

```
% c:/Users/Legion67/Documents/Prolog/depth_first_p.pl compiled 0.00 sec, 33 clauses
?- start.
[up,up,up,up,up,up,up,up,up,right,down,down,down,down,down,right,up,up,up,up,up,right,down,down,down,
down,down,right,up,up,up,up,up,right,right,down,down,right,up,up,right,down,down,right,down,down,down
,left,up,left|_10568]
true
```

Figure 2. Maze 2

Iterative deepening Search

This acts very similar to the depth first search. The only difference is it takes the depth into account. It goes to one level tried all the actions at that level then moves to the next depth level.

```
start:-
    id(S).

 id(Sol):-
   maxDepth(D),
   startPosition(S),
   length(_, L),
   L =< D,
   id_search(S, Sol, [S], L),
   write(Sol).

 % id_search predicate provides the ID search.

 id_search(S, [], _, _):-
   finalPosition(S).
 id_search(S, [Action|OtherActions], VisitedNodes, N):-
   N>0,
   allowed(Action, S),
   move(Action, S, NewS),
   not(member(NewS, VisitedNodes)),
   N1 is N-1,
   id_search(NewS, OtherActions, [NewS|VisitedNodes], N1).
```

Solutions:

```
% c:/Users/Legion67/Documents/Prolog/iterative_deepening.pl compiled 0.02 sec, 23 clauses
?- start.
[down,down,left,left,up,up,up,up]
true .

?-
```

Figure 3. Maze 1

```
% c:/Users/Legion67/Documents/Prolog/iterative_deepening.pl compiled 0.00 sec, 34 clauses
?- start.
[up,up,up,up,up,up,up,up,up,right,right,right,right,right,right,down,down,right,right,right,
down,down,left,left]
true .

?-
```

Figure 4. Maze 2

A* Search

The A* implementation consists of 3 main parts.
Astar: starts the search and fills a list with moves to reach the goal.
Astar_search: these are the predicates that implements the search. It takes 3 parameters, the first is a list of nodes that are at the front of the current path, the second is the list of positions that have already been visited, the third is the list of actions that lead to the solution.
Generatechildren: generates the children of a particular position, checks all the allowed actions (moves) for that position.

```
start:-
    astar(S).

astar(Solution) :-
    startPosition(S),
    heuristic(S, _, HeuristicS),
    astar_search([node(S, [], 0, HeuristicS)], [], Solution),
    write(Solution).


astar_search([node(S, ActionsListForS, _, _)|_], _, ActionsListForS):-
    finalPosition(S).

astar_search([node(S, ActionsListForS, ActualPathCost, HeuristicCost)|Frontier], ExpandedNodes, Solution):-
    findall(Az, allowed(Az, S), AllowedActionsList),
    generateChildren(node(S, ActionsListForS, ActualPathCost, HeuristicCost), AllowedActionsList, ExpandedNodes, SChilderenList),
    append(SChilderenList, Frontier, NewFrontier),
    predsort(a_star_comparator, NewFrontier, OrderedResult),
    astar_search(OrderedResult, [S|ExpandedNodes], Solution).

generateChildren(_, [], _, []).
generateChildren(node(S, ActionsListForS, PathCostForS, HeuristicOfS),
                 [Action|OtherActions],
                 ExpandedNodes,
                 [node(NewS, ActionsListForNewS, PathCostForNewS, HeuristicCostForNewS)|OtherChildren]):-
    move(Action, S, NewS),
    not(member(NewS, ExpandedNodes)),
    cost(S, NewS, Cost),
    PathCostForNewS is PathCostForS + Cost,
    heuristic(NewS, _, HeuristicCostForNewS),
    append(ActionsListForS, [Action], ActionsListForNewS),
    generateChildren(node(S, ActionsListForS, PathCostForS, HeuristicOfS), OtherActions, ExpandedNodes, OtherChildren),
    !.

  % Used to backtrack on any other possible action in case of fail.

generateChildren(Node, [_|OtherActions], ExpandedNodes, ChildNodesList) :-
    generateChildren(Node, OtherActions, ExpandedNodes, ChildNodesList),
    !.
```

Solutions:

```
% c:/Users/Legion67/Documents/Prolog/a_star_p.pl compiled 0.00 sec, 28 clauses
?- start.
[down,down,left,left,up,up,up,up]
true .

?-
```

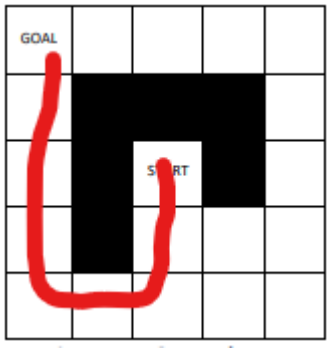
Figure 5. Maze 1

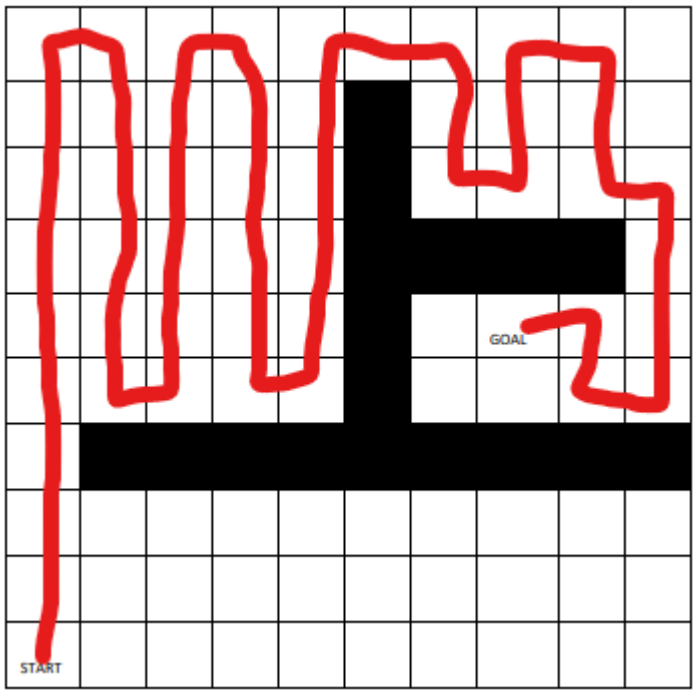
```
% c:/users/legion67/documents/prolog/a_star_p compiled 0.00 sec, 11 clauses
?- start.
[up,up,up,up,up,right,right,right,right,up,up,up,up,right,right,down,down,right,right,right,
down,down,left,left]
true
```

Figure 6. Maze 2

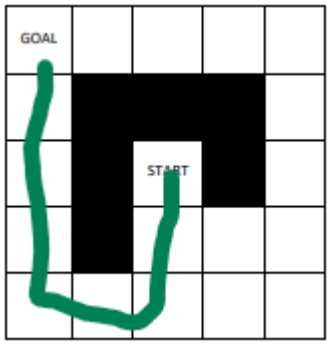
Paths taken

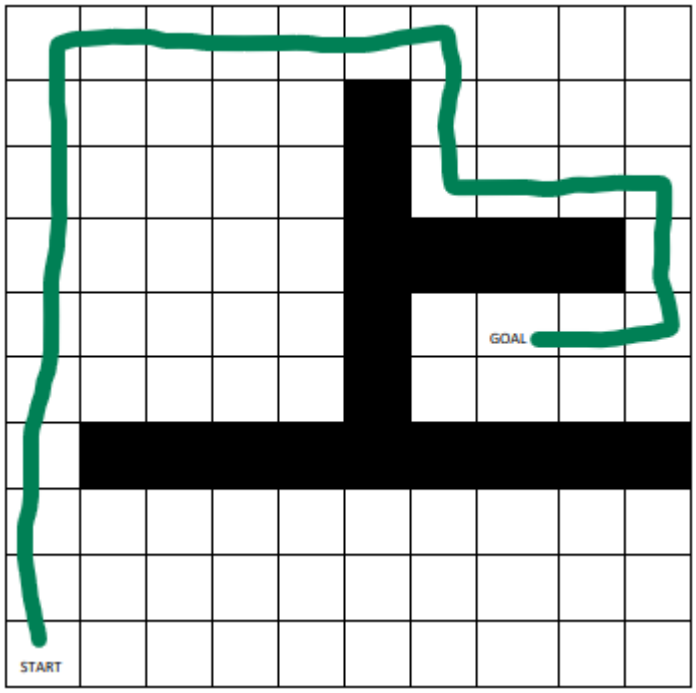
Depth-First:





Iterative Deepening:





A*: