**Artificial Intelligence**

**Submission – Assignment 4**

Mahathi Priya Appini (110162023)

Shriya Gupta (110168559)

**Translating the Maze into Graphical Form for Prolog-Based Algorithms**

We represented the maze as a graph in the memory by numbering the nodes in the maze map and identifying the nodes that were connected through walls. For this, we used the following pseudocode –

1. Get the walls in the game state.
2. Get the Pacman position and mark it as the start state.
3. Identify the goal node as the termination node.
4. Check to see that there is food in the maze. If not, exit.
5. For node in current state, identify the first child node.
6. If a wall exists, that connects the current node to the child node, insert this as a connected fact in the prolog maze file.

The format of this rule would be *connected (parentnode, childnode, direction)*.

1. Check for next child node unless four children have been checked. Go to step 6.
2. Go to next node in the maze unless end of maze is reached.

This algorithm resulted in the creation of prolog facts for the different mazes identified in the project.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | **5** |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| **21** | 22 | 23 | 24 | 25 |

Figure 1. Graphical Representation of Maze

Here, we have used peach color to represent the edges between the nodes and the color red to highlight the start node at 5 and the goal node at 21 has been represented in green.

For our purposes, we declared a recursive predicate, member, as follows –

member(X,[X|\_]).

member(X,[\_|Tail]):- member(X,Tail).

We also declared another predicate, append, as follows. It stores the result of append operation in L3 –

append([],L,L).

append([H|T],L2,[H|L3]) :- append(T,L2,L3).  
  
Solution 1.   
  
 **Pseudocode for DFS -**   
  
dfs(S,Path,Path) :- goal(S).

dfs(S,SoFar,Path) :- arc(S,S2),

\+(member(S2,SoFar)),

dfs(S2,[S2|SoFar], Path).  
  
  
Here, we have used the startnode (originail Pacman position) as S, the Visited List has been tracked in the Path, and the Directions have been tracked along the path that the Pacman travels.

To execute, enter the following in command line:   
 *For Mini Maze -*  
python.exe pacman.py -l tinyMaze -p SearchAgent

*For Medium Maze -*  
python.exe pacman.py -l mediumMaze -p SearchAgent

*For Big Maze -*  
python.exe pacman.py -l bigMaze -z .5 -p SearchAgent  
 **Results -**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Maze** | **Path Cost** | **Time Taken** | **Search Nodes Expanded** | **Outcome** | **Avg. Score** | **Scores** | **Win Rate** | **Record** |
| Mini Maze | 8 | 1.7s | 18 | Victorious | 502 | 502 | 1/1 | Win |
| Medium Maze | 68 | 1.7s | 282 | Victorious | 442 | 442 | 1/1 | Win |
| Big Maze | 210 | 1.8s | 648 | Victorious | 300 | 300 | 1/1 | Win |

Solution 2.   
  
**Pseudocode for BFS –**

doBFS(Start, Direction) :- bfs([[Start]],Direction).

/\* found a goal node in BFS. The goal node is always at head of queue. We are comparing the cell number\*/

/\* each node in queue(list) consists of <cell No, direction> \*/

bfs([[Cell|Node]|\_], [Cell|Node]) :- goal(Cell,\_).

bfs([FirstNode|RestNodes], Direction) :-

/\* get the first node of queue and generate successors \*/

findAll(FirstNode, Successors),

/\* append the successors of this element to queue. the First node is removed from queue \*/ append(RestNodes, Successors, Queue),

/\* evaluate BFS recursively on this queue \*/

bfs(Queue, Direction).

/\* get the successors of a cell in that node and store in Successors that follows a template \*/

findAll([Cell|Node], Successors):-

setof([NextNode, Dir, Cell|Node],

(connected(Cell, NextNode, Dir), not(member(NextNode,[Cell|Node])) ),

Successors),

!.

findAll(Node, []).  
  
To execute, enter the following in command line:   
 *For Mini Maze -*  
python.exe pacman.py -l tinymaze -z .5 -p SearchAgent -a fn=bfs

*For Medium Maze -*  
python.exe pacman.py -l mediumMaze -z .5 -p SearchAgent -a fn=bfs

*For Big Maze -*  
python.exe pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=bfs  
 **Results -**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Maze** | **Path Cost** | **Time Taken** | **Search Nodes Expanded** | **Outcome** | **Avg. Score** | **Scores** | **Win Rate** | **Record** |
| Mini Maze | 8 | 1.1s | 17 | Victorious | 502 | 502 | 1/1 | Win |
| Medium Maze | 68 | 1.1s | 281 | Victorious | 442 | 442 | 1/1 | Win |
| Big Maze | 210 | 1.2s | 647 | Victorious | 300 | 300 | 1/1 | Win |

Solution 3.

**Pseudocode for ASTAR –**

 astar(Path) :- start(S), astar(S,Path).

astar(S,[S]) :- goal(S), !.

 astar(S,[S|Path]) :- findall(Dist-SS, (arc(S,SS), h(SS,Dist)), L), keysort(L,SortedL),

member(\_-NextS,SortedL),

astar(NextS,Path).

Here, we have modelled the heuristics of Manhattan Distance as facts in a new file, and sorted them on keysort.

To execute, enter the following in command line:   
 *For Mini Maze -*  
python.exe pacman.py -l tinyMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

*For Medium Maze -*  
python.exe mediumMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

*For Big Maze -*  
python.exe pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic  
 **Results -**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Maze** | **Path Cost** | **Time Taken** | **Search Nodes Expanded** | **Outcome** | **Avg. Score** | **Scores** | **Win Rate** | **Record** |
| Mini Maze | 8 | 1.5s | 17 | Victorious | 502 | 502 | 1/1 | Win |
| Medium Maze | 68 | 1.5s | 281 | Victorious | 442 | 442 | 1/1 | Win |
| Big Maze | 210 | 1.6s | 647 | Victorious | 300 | 300 | 1/1 | Win |

**Observations on Solution 1, 2 and 3 –**

* DFS is taking longer than BFS for each maze.
* Pacman is able to emerge victorious with all algorithms, with similar number of nodes expanded and score achieved.
* With increase in maze size, as input nodes increases, the cost of traversal increases until solution can be reached and therefore, time taken increases.
* A-Star is faster than DFS but is taking longer than BFS – this could be due to the additional overhead of computing heuristic, but AStar may end up overperforming BFS with the maze size growing larger.

Solution 4.

**Pseudocode for Four corners –**

1. Start node = Pacman Original Position, Goal Node = First corner
2. Implement BFS until Goal Node is reached
3. Update start node as goal node that was reached
4. Reduce count of goal nodes to be reached
5. Update goal node as next goal node
6. Go to step 2 until all 4 goal nodes are found
7. Exit

**Output -**

C:\Python27\python.exe C:/Users/mahathi/Documents/artificial\_intelligence/Assignment4/search/pacman.py -l tinyCorners -p SearchAgent -a fn=BreadthFirstSearchCorners,prob=CornersProblem

Path found with total cost of 28 in 5.5 seconds

Search nodes expanded: 228

Pacman emerges victorious! Score: 512

Average Score: 512.0

Scores: 512

Win Rate: 1/1 (1.00)

Record: Win

C:\Python27\python.exe C:/Users/mahathi/Documents/artificial\_intelligence/Assignment4/search/pacman.py -l mediumCorners -p SearchAgent -a fn=BreadthFirstSearchCorners,prob=CornersProblem

Path found with total cost of 107 in 6.4 seconds

Search nodes expanded: 4739

Pacman emerges victorious! Score: 433

Average Score: 433.0

Scores: 433

Win Rate: 1/1 (1.00)

Record: Win

**Conclusions & Results –**

Average execution time over all three mazes is least for BFS, followed by A-Star and DFS – This suggests that BFS may be optimal solution. However, the additional cost complexity of calculating the heuristic for A-star may have slowed it down, but the benefits of heuristics search may outweigh the cost in the long run, making it the optimum solution. For these mazes and this particular problem, BFS works best.

With the implementation of four corners over Breadth-First Search, Pacman emerged victorious within reasonable amount of time and this reasserted the applicability of this algorithm.

Challenges & learnings – Prolog is a declarative programming language. It was challenging to adapt to this style of writing code where logic is expressed in relations, which themselves are represented by facts and rules. We are now better at dealing with predicate logic and horn clauses. Moreover, there are certain differences between XSB Prolog and SWI prolog – all functionality is not supported by both and the implementation is more nuanced. Moreover, to account for slow machine response, we made sure that we did not revisit already expanded nodes – otherwise big maze could not be solved.

**References –**

* Class slides on Prolog
* xsb.sourceforge.net
* www.csee.umbc.edu/courses/771/current/presentations/prolog%20search.pdf
* www.learnprolognow.org/lpnpage.php?pagetype=html&pageid=lpn-htmlse24
* www.cs.nott.ac.uk/~bsl/G52APT/slides/10-Breadth-first-search.pdf