Foundation of Intelligent Systems

Project 1 – Report

Abhiram Ravi Bharadwaj
ab8136@rit.edu

Contents

File Descriptions

File Name	Description
node.py	The file that contains the basic node for the tree structure of IDS and A*. This is common to both search algorithms.
ids.py	The implementation of IDS is in this file. Import this to any problem to use. The main function to be called is *ids*.
a_star.py	The implementation of A* is in this file. Import this to any problem to use. The main function to be called is *a_star*.
maze_solver.py	The problem for maze is in this file. The functions in this file are specific to the maze problem. This file is executable and takes as a command line argument the file path of the input maze.
draw.py	The turtle drawing is present in this file. Import this file and call the *draw_maze* function to draw the maze with the path to be taken.
n_queens.py	This file is similar to the maze.py file. This is the problem file for the n-queens. This file is executable and takes as command line argument the number of queens to be solved for.
maze	This is a sample dataset that can be used to test all the search algorithms. For larger mazes, the search algorithms take considerably longer time. This maze is of size 10x20.

Problem Statement

In this project, we were required to implement the search algorithms IDS (Iterative Deepening Search) and A* to find a solution to a given maze. For A* we had to use four heuristics. We had to analyze the performance of these algorithms based on performance metrics that we find suitable. It was given that the start location of the maze would be the top left cell and the exit location would be the bottom right cell.

For a small bonus, we could show a diagrammatic solution using turtle. For a larger bonus, we could solve another problem like n-queens or rubik's cube using the same implementation.

State Space Description

States: For the maze problem, we can assume that there is a traveler who wants to go from the start of the maze to the end of the maze. The states for the maze problem computes of all the various positions in the maze that the traveler can be in.

Initial State: Initially, the traveler is at the top left cell of the maze. The maze has certain cells where the traveler cannot go to.

Actions: The actions are all the possible directions the traveler can go. In this case it would be right, up, left and down.

Transition Function: The transition function returns the new position of the traveler after performing the action. This would be the new state.
Goal State: In the goal state, the traveler has successfully found a path to the exit. This translates that the traveler is currently at the bottom right cell of the maze.
Path Cost: All actions in this problem have the same step cost of 1. The total path cost would be the number of steps taken by the traveler in reaching the destination.

State Space Representation

The representation details mentioned below states one possible way of addressing the problem. The motivation behind this approach is to be able to use the IDS and A* algorithms as a macro. It should be possible to just import these algorithms and solve any problem using these search methods.
In this approach, the state space description is stored in a node. The node here can be thought of as a node in the tree. Each action performed on a node creates a branch from the node leading on to a child node. Each node stores the current position, the parent node, the action that led to the current node and a list of its children.
Each of IDS and A* takes as input an initial node, a list of actions, an action function and a goal test function. At each step, actions are performed on the node and feasible ones are considered. Each of the chosen node is passed to the goal test function to determine when the algorithm should stop. The parameters passed to these search algorithms are problem specific and would be defined in the problem. This way, the search algorithms just call the functions as and when needed.

Heuristics

A heuristic is admissible if it never overestimates the cost of reaching the goal. A heuristic is consistent if for all nodes, the estimate is higher than the estimates of its children. A heuristic that is admissible and consistent is a good heuristic. For this project, four heuristics are used.

1. **Random heuristics:** This heuristic returns a random number in the range of 0-10000. This is not an admissible heuristic because we cannot guarantee that it is optimistic.
2. **Zero heuristics:** This heuristic returns zero for all nodes. This does not benefit the algorithm as it makes the algorithm into a plain and simple breadth first search algorithm. Though this heuristic guarantees optimism and hence is admissible.
3. **Manhattan distance:** This heuristic returns the manhattan distance or the taxi cab distance. At each step, the next node will always have a heuristic value lesser than the parent. Hence it is consistent. Also, the total estimate can never be more than the actual cost. Hence it is admissible.
4. **Straight line distance:** In this heuristic, for each node, we consider its straight line distance from the destination. This guarantees that the next step will always be closer to the destination than the current step. Also, the actual distance can never be more than the straight line distance. This heuristic is thus admissible and consistent and hence a good heuristic.

Performance Metrics

The performance of search algorithms can be measured based on a combination of the time taken, optimality and also the branching factor. For the maze problem, the optimality can be measured in terms of the number of steps taken to reach the goal. A search algorithm that is the fastest, takes least number of steps and has the least branching factor is the best search algorithm. For the maze problem, it happens to be the A* search with the manhatten heuristic which satisfies all these conditions and trumps the other options.

In case of IDS, it gets really slow once a certain depth has been reached. In this case, the observation was that it got slow after around 15 iterations. This problem does not exist with the A* search algorithm.

Note: The runtime of A* for maze problems of medium size starts to get slow with the current implementation. This is attributed to the use of a deepcopy mechanism to generate the child states.

The following table shows the comparison of the algorithms for a maze of size 10 x 20:

Algorithm	**Time** (seconds)	**Steps**	**Branching Factor**
IDS	2.43	31	1.27
A* (random)	0.14	49	1.12
A* (zero)	0.20	29	1.22
A* (manhattan distance)	0.02	29	1.12
A* (straight line distance)	0.11	29	1.20

The following table shows the comparison of the algorithms for a maze of size 40 x 40:

Algorithm	**Time** (seconds)	**Steps**	**Branching Factor**
IDS	N.A.	N.A.	N.A.
A* (random)	5.47	115	1.06
A* (zero)	7.86	79	1.10
A* (manhattan distance)	0.96	79	1.07
A* (straight line distance)	4.43	79	1.09

Bonus Questions

Of the attached files, the *draw.py* file handles drawing the maze with the path taken. The screenshots of some of the solutions can be found below. This file also draws out the chessboard for the n-queens problem.

Another attached file is the *n_queens.py* which calls the IDS and A* search algorithms to solve the n-queens problem. There are no modifications that need to be done to IDS or A* to make it compatible with the n-queens. The n-queens has its own set of actions, action function and goal test function that is passed for these search algorithms to compute the final state. To test for proper functional.

Screenshots

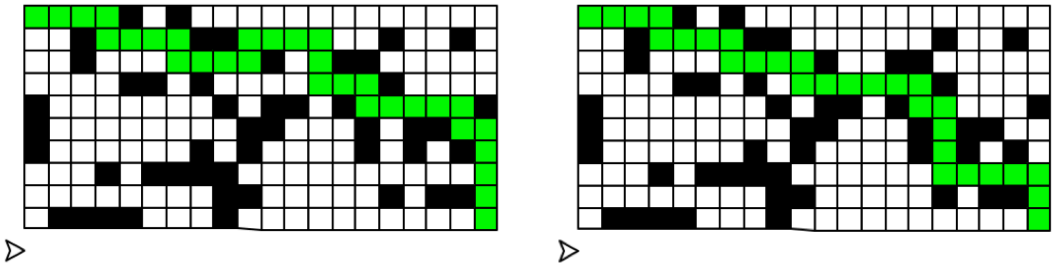

Figure 1 10x20 Maze
Left: IDS
Right: A with Straight Line Heuristic*

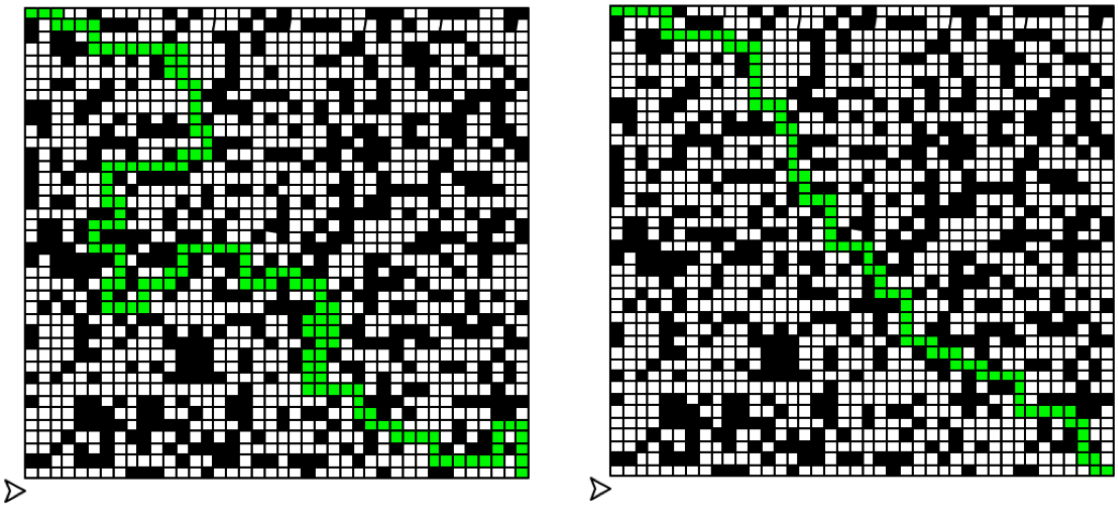

Figure 2 40x40 Maze
Left: A with Random Heuristic*
Right: A with Zero Heuristic*

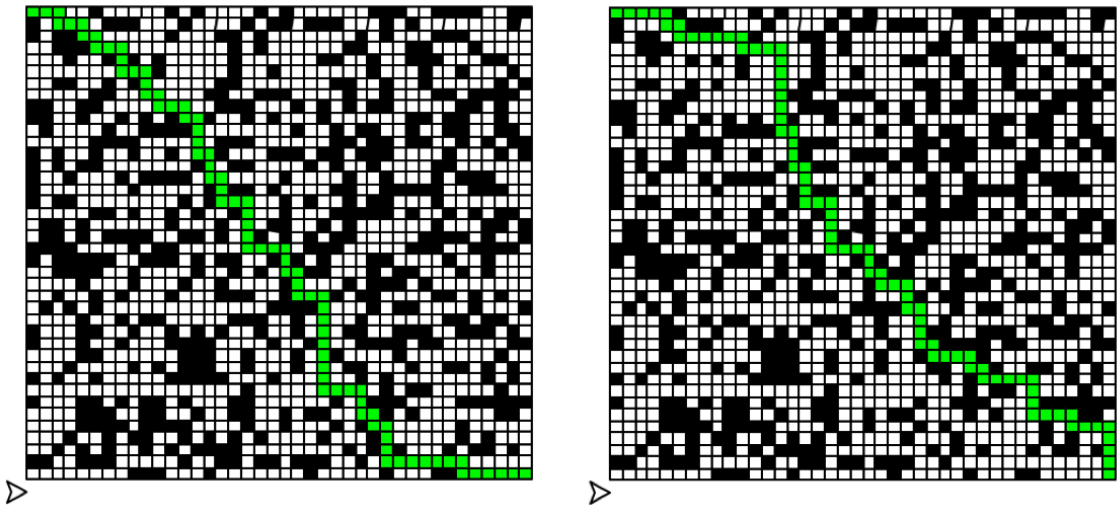

Figure 3 40x40 Maze
Left: A with Manhattan Distance Heuristic*
Right: A with Straight Line Distance Heuristic*

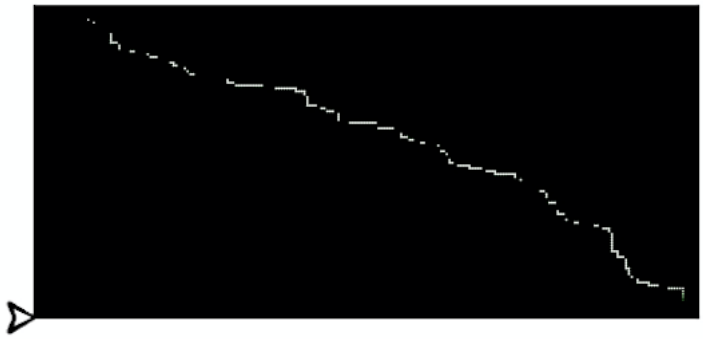

Figure 4 200x400 Maze

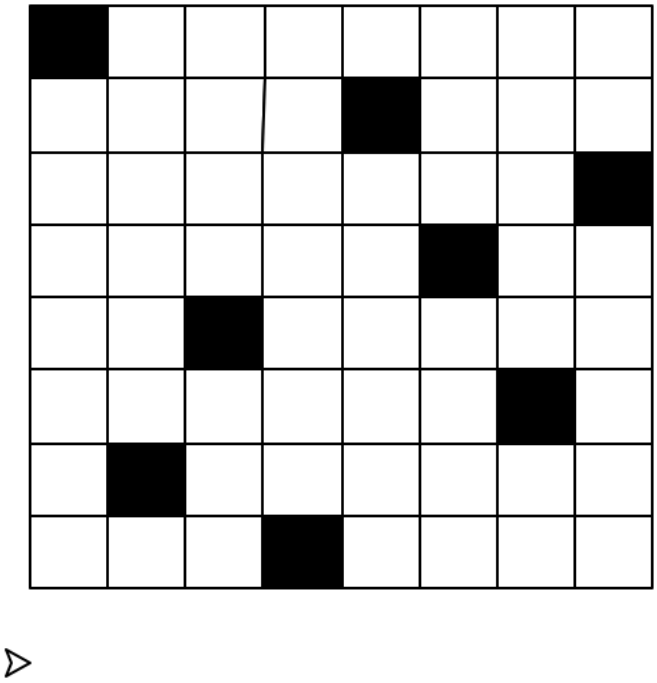

Figure 5 n queens with 8 queens