Foundation of Intelligent Systems
Project 1

Manasi Bharat Gund
February 11, 2019

Problem Statement

Implement IDS and A* using four heuristics to a maze-solving program. Profile the code and determine how well each approach works.

State Space Search Representation for the maze-solver

- **States**: States will be different representations of the maze, w.r.t different current location in the maze (For the memory saving purpose, it is only represented as the current location's coordinates in the program)
- **Initial state**: The state with the current location as the starting location (Here it's (0, 0))
- **Actions**: go right, go down, go left, go up (move one block in any of the four directions)
- **Transition model**: From the current state, on performing the possible action, it goes to the next state with the updated location. For example,

  problem.result(state with (0, 0), go_right) = state with (0, 1) if the action is possible, i.e. the digit representing that location (if present) is 0

- **Goal test**: Reaching the exit of the maze. Here, with location (last row, last column)

- **Path cost**: Step cost for moving one square is one. Path cost will be the total cost from the initial state to that state (Addition of step costs)

Problem Design

- A **MazeState** class represents the states. Instances of MazeState class are created to create a new state.
  This class represents the state of the problem with the current location coordinates in the maze (Attributes: current_row, current_column)
- A **Node** class is a generic class built to maintain a data structure to keep track of the tree. Each node is described with these four components:
  1. State – An instance of MazeState. Two nodes can have the same state, but they might be reached through different path or action;
  2. Parent – Parent Node that generated this node;
  3. Action – The action that was taken to reach this state.
  4. Path-cost – Total cost so far to reach here, g(n), the path from the node with initial state to this node. Here, step cost is one, so number of steps taken will be the path cost.

  This class helps the program achieve two things:
  1. The tree structure is maintained
  2. The solution() function uses the parent pointers to trace back the path, if present.

- Class **Problem** represents the problem based on the state space search representation of the problem. This could be generic, define any problem with:
  1. Initial State – This is just an instance of MazeState class in the maze-solver (to save memory, just the row and column coordinates. Thus, I have maintained an additional attribute: Maze, a matrix of all squares of the maze.)
  2. Set of actions – as in the state space description
  3. Goal – coordinates of the exit point in the maze. In this case, list: goal[last row, last column].

State Space Description

Given a problem maze, the Problem class instance is formulated by parsing it, and extracting the components needed to define the problem:

the initial state, the maze of 0s and 1s (1s are the obstacles), goal coordinates. This problem instance is passed as a parameter to a search algorithm. IDS and A* are two search algorithms implemented in the program. Both the algorithms use the problem object passed to check the goal test, actions.

Iterative Deepening Search

IDS takes the generic problem object and searches for the path, with limited depth starting from 0. A parent node (instance of Node class) is created starting from the node with the initial state. IDS then goes on checking the goal test for every node created on every depth, till the depth limit is reached. For the parent node, if not the solution, it checks for the actions possible using the problem.actions() function, creates Node instances for a state resulting from that action, and then goes on recursively checking for the child nodes.

As the depth went on increasing, IDS proved to be an inadequate algorithm even for the small mazes provided (Ran okay till depth 15-16.) The maze problem squares were bidirectional, making this problem a graph. IDS is mostly suitable for tree search, so maintaining a path of the particular branch that is being explored at that moment could be the solution to make this problem act like a tree, which defies the purpose of using IDS over BFS, DFS to save memory. Otherwise, after a particular depth, it keeps on exploring the same steps multiple times, making it excruciatingly slow. For a maze with the solution on depth 29, my program took 5 felt-like-never-ending hours.

IDS implemented is a generic function, that given any other problem instance of that specific problem class, say N-queens, will still return the solution, as it uses the problem instance passed to check actions, results, goal test etc.

A* search

This search algorithm takes one extra parameter with the generic problem instance: heuristic function. A* maintains a priority queue to store the nodes created, by prioritizing them by their f(n) = path cost g(n) + heuristic value h(n). So, the efficiency

of a good A* search function depends on a good heuristic value. Let's analyze the different heuristics we have used for this problem:

1. **Heuristic that returns 0**: Since h(n) for any node is always zero, the efficiency of f(n) depends only on the path cost. Thus, A* with this heuristic will act like Dijkstra's, or BFS with the uniform cost. The heuristic in this case is still optimistic.
2. **Heuristic that returns random value (till 999)**: The randomness makes the h(n) inconsistent and at times, not optimistic. This heavily affects the search. With a very large h(n) value all the time, the search will depend heavily on h(n) and not on g(n), and A* will be Greedy Best-First-Search. This search almost always takes a greater number of steps to reach the solution that the search with the other three heuristics.
3. **Heuristic with Manhattan distance**: This heuristic function returns the square-wise distance required to reach the goal. Since it considers each square, and returns the addition of horizontal and vertical distance, it never overestimates the cost, and is always consistent and optimistic, making it admissible. This is an example of a **Good Heuristic**.
4. **Heuristic with Euclidean Distance**: h(n) is the straight-line distance from the state of that node to the goal. Again, an **Admissible Heuristic**, that never overestimates, is consistent and optimistic.

The path given by the last two heuristics will be the optimal solution for this problem, because of the good heuristics used. The path with the h(n) = 0, was also mostly good. The path with random h(n) performed evidently bad with different mazes.

A* function I have written is a generic function: given any other problem instance of that specific problem, say N-queens, class and passing any heuristic function will return the solution.

Both these algorithms on passing the goal test calls solution() function that traces back the path using Node structure, and returns it in a form of easily accessible list.

In the outer layer, after calling the search algorithm and getting the solution path from it, the solution (if any) can be printed on the console, as well as with turtle (for now, called on A* with Euclidian distance heuristic.)

Performance Metrics:

Performance of a good algorithm for this problem can be decided by the capability of an algorithm to get an optimal and complete solution in a reasonable time. While IDS gave the best path for the smaller mazes, as the maze size went on increasing, IDS took hours to even get to the solution. A* gives the best solution (does not necessarily mean the shortest) within a reasonable time. Like, for game playing environments, time required to get to the solution is a major factor to consider. A* managed to get the solution with a good time-steps trade-off. The branching factor for A* was better than IDS, as the maze size increased, As it increased further, IDS took a really long time to even record the observations. In A* search, for good heuristics, like h4, the number of steps required to reach the solution was better than bad heuristics, like h2.

So, for this problem, a good Performance Measure will be a complete and optimal solution with the balanced trade-off between the number of steps, branching factor, time taken to reach the solution. My observations are recorded in the table below.

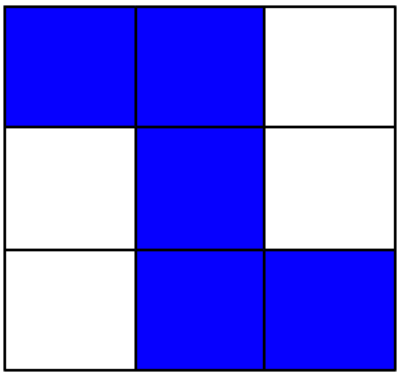
Figure 1. Comparison of the search costs and effective branching factors for the IDS and A* algorithms with different heuristics.

Maze size	**Nodes generated | Steps to solution**					**Effective Branching Factor**				
	IDS	**A***				**IDS**	**A***			
		h1	**h2**	**h3**	**h4**		**h1**	**h2**	**h3**	**h4**
3x3	29|5	18|5	11|5	18|5	17|5	1.76	1.78	1.61	1.78	1.76
5x5	725|8	38|9	30|9	37|9	36|9	2.07	1.49	1.45	1.49	1.48
10x20	-	408|29	291|37	281|29	320|29	-	1.23	1.16	1.21	1.22
40x40	-	2246 | 79	1306 | 101	832 | 79	1453 | 79	-	1.10	1.12	1.08	1.09
200x400	-	178028 | 599		72276 |	162405 | 599	-	1.021			1.020

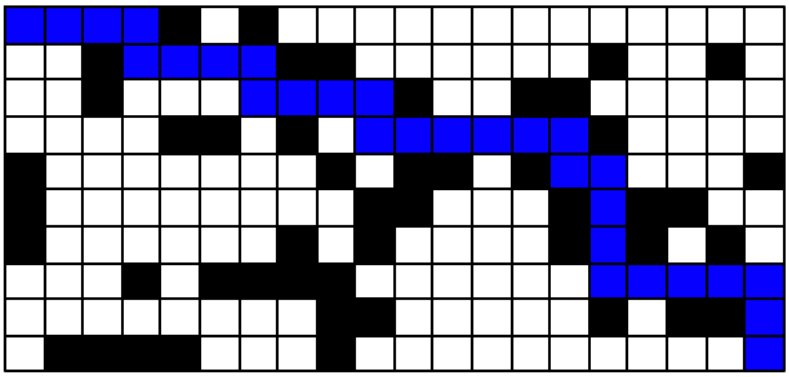
Maze solutions with turtle

These are the screenshots of solution for different mazes with A* search using Heuristic_4

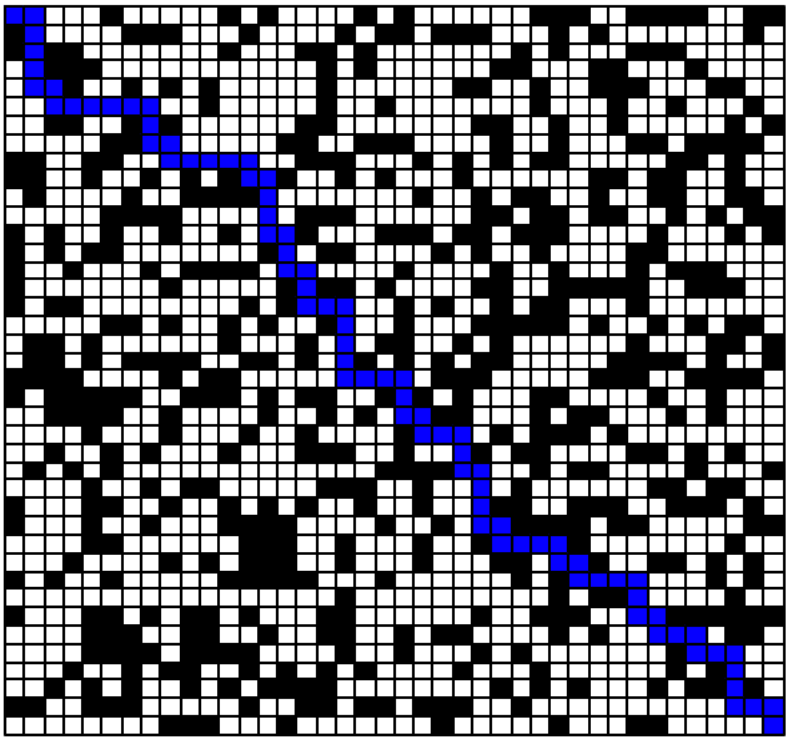
3 x 3 Maze



10 x 20 Maze



40 x 40 Maze



N-QUEENS PROBLEM

The N-queens problem is implemented(nqueens.py) using the generic classes and functions. Only specific NqProblem class and NqState class was written for the n-queens problem.

The command line argument is the size of the chess board. For example, input 4 gives a 4 x 4 chessboard.

Turtle maze for a 8 x 8 board: