CS 331 - Spring 2016 - Implementation 1

Cody Malick - Andrew Tolvstad
malickc@oregonstate.edu, tolvstaa@oregonstate.edu

April 15, 2016

1 Implementation Assignment 1

1.1 Methodology

1.1.1 Breadth-First Search

Getting this algorithm working was quite a bit of trial and error, as fully expanding all possible nodes until a solution is found is memory intensive. We implemented this algorithm using queue for the general data structure, we used a set data structure containing tuples representing individual states visited, and a trace function "ascend()" to return the path the algorithm takes to find the solution. The signature vector was used across all algorithms.

1.1.2 Depth-First Search

Depth-first was much faster than BFS. The only difference between this and BFS implementation was that we used a stack for our data structure.

1.1.3 Iterative Deepening Depth-First Search

This is identical to DFS, with the added element of a maximum depth variable. Through some experimentation, we found that starting depth at one, then doubling it each time was a good way to increase the depth quickly enough. We considered doing an additive approach, but it was too slow as the data set size increased.

1.1.4 A-Star Search

For A-Star we used a priority queue, a self-sorting queue, that sorts based on the heuristic we assigned it. In this case, the heuristic we used was absolute distance from the current state, to the end state. We defined this by subtracting the absolute value of how many people still need to be moved to the goal shore versus the goal state.

Here's pseudo code on this idea:

```
abs((current: goal_shore_population)−(goal: goal_shore_population)) +
abs((current: source_shore_population)−(goal: source_shore_population));
```

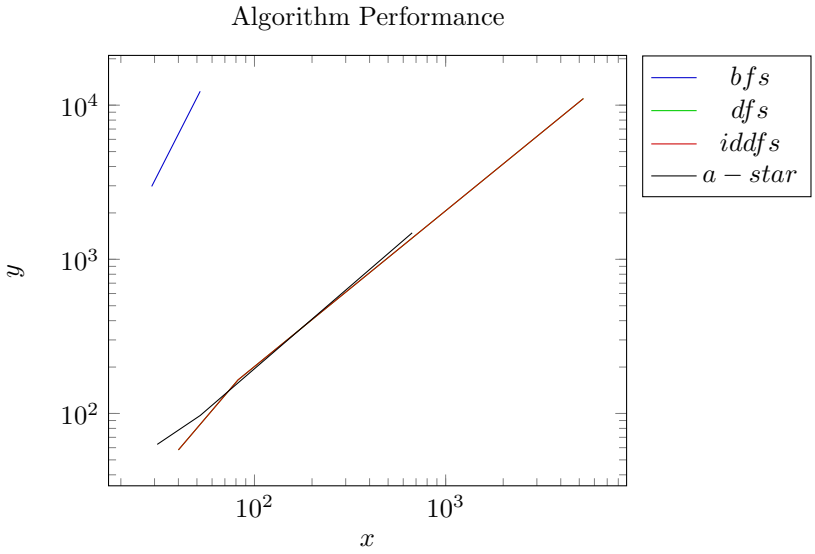
1.2 Results

Here are the results of our implmentations:

	Test Case 1	Test Case 2	Test Case 3
BFS	Path:29 Nodes:2971	Path:52 Nodes:12305	Path: - Nodes: -
DFS	Path:40 Nodes:58	Path:82 Nodes:165	Path:5256 Nodes:11040
IDDFS	Path:40 Nodes:58	Path:82 Nodes:165	Path:5256 Nodes:11040
A-Star	Path:31 Nodes:63	Path:52 Nodes:97	Path:668 Nodes:1485

Here is a graph illustrating the results of our executions. We used a log x and y axis to appropriately show the differences in performance. The x axis is solution size, and the y axis is number of nodes traversed:

1.2.1 graph



1.3 Discussion

Overall the results for breadth-first were expected. BFS is very inefficient. After implementing the signature vector, our implmentation had a much easier time finding the solution on smaller data sets, eliminating repeated states.

This still only reduced the problem size slightly. We were still iterating over an enormous tree to find the solution.

We had quite a time running this algorithm against test case three, containing eighty-eight missionaries and eighty cannibles. Before sorting out a few memory leaks (we implmented all our algorithms in C++), we were hitting thirty-plus gigabytes of RAM while BFS was trying to resolve the solution. After further investigation, the memory issue was due to the overhead in the class system written, and the size of the signature vector we used.

Depth-first was quite a different experience from BFS. Depth-first was able to resolve the solutions for all three test cases quickly and efficiently, contrary to what we believed would happen. We thought it would be as slow or slightly faster, but it ended up being drastically better in performance and number of steps taken.

Iterative Deepening Depth-First search we expected to perform on par with or better than DFS, but it ended being exactly the same. This was a little surprising, but actually made sense given the size of the data sets we have. We would only expect a large performance increase given an extremely large set of data, where solutions exist at many different points relative to this data size.

A-Star we expected would perform better than all the above, given the priority queue and heuristic. With the priority queue sorting the possible paths in order of the heuristic, it could find solutions more efficiently in different positions that the other algorithms do not have the capacity to find.

1.4 Conclusion

From these results, we can clearly see that A-Star is the most efficient algorithm for this setup. This was largely expected as it's the only one that had any heuristic and path prioritization.