Benchmarking Path Search Algorithms

Amey Darekar, Konstantin Tenman

Institute of Computer Science, University of Tartu

Introduction

The major goal of this project is to revisit popular path-finding and graph traversal algorithms in order to determine how well they perform when applied to the sliding puzzle problem.
In this project we implemented three search algorithms: breadth-first, depth-first, and A* search algorithms. We compared these path finding techniques in terms of their efficiency and speed, as well as set a baseline for their performance [1].

Sliding Puzzle Problem

A sliding block puzzle is a type of combination problem in which neighbouring blocks or tiles are slid into the empty space to achieve a certain goal pattern or configuration. In our example problem we use a simple numbered tile sliding problem, with zero representing the blank tile. The goal state, as seen in Figure 1, is arranged as so that all tiles should be arranged in increasing order, with tile one at the top of left corner of the board and two next to it, and so on until the end. The blank tile slot goes to the last place, in the bottom right end of the board.
With every move (sliding a tile into empty slot), we generate a new state of the board. Each such state leads to several new states (between 2 to 4). We can map these connected states as a graph by representing each state as a node. Using the graph search algorithms, we can look for a path from the initial state to the goal state.

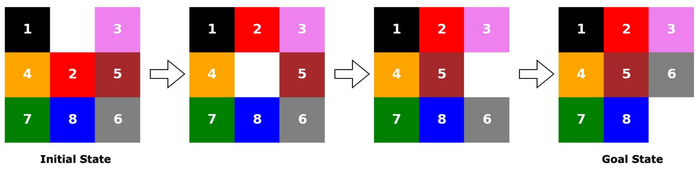

Fig. 1: Example solution

Breadth-first Search

The breadth-first search (BFS) algorithm is used to search a tree data structure for a node that meets a set of criteria. It begins at the root node and investigates all nodes at the current depth level before moving on to nodes at the next depth level [2]. More memory, typically a queue, is required to keep record of the child nodes that have been discovered but not yet visited.

Depth-first Search

The depth-first search (DFS) algorithm is used to explore and search data structures such as trees and graphs [4]. The algorithm starts at the root node (in the case of a graph, any random node can be used as the root node). Unlike Breadth-first search which evaluates all possible neighbors before moving to next paths, DFS evaluates each path as far as feasible before returning. DFS uses Stack data-structures for performing search.

A* Search

A-star is an informed graph traversal and path search method which uses of heuristic function to determine current and neighboring state's distance from goal state [3]. A* selects its path such that it reduces the value of heuristic function. Due to its completeness, optimality, and efficiency, that is widely applied in many domains of computer science. Its $O(b^d)$ space complexity is one of its primary practical drawbacks, as it saves all created nodes in memory. We will calculate our heuristic function as Manhattan distance for each tile for current state with each tile of the goal state.

Results

We generated 50 problems by creating random permutations for 2x3 and 3x3 puzzles. We recorded how many states were visited for each algorithm and time taken as each path was being expanded. We observed that significant number of problems were impossible to solve as we cannot swap any tiles, so we opted to benchmark both solvable and unsolvable problems independently and report a comparative analysis. Following data shows that BFS performed better compared to DFS. A* performed at par with BFS in smaller problems and outperformed both BFS and DFS for 3x3 solvable problems.

Algorithm	Solvable	
	2x3	3x3
BFS	218	111060
DFS	187	144198
A*	119	8228

Fig. 2: Average States visited

As observed in Figure 2, for small solvable problems, BFS search is the least efficient, whereas A* search is the most efficient. BFS visits 218 states on average, while A* searches 119. If the problem size is expanded by 1.5 times (2x3 -> 3x3), A* search visits on average 8228/119=69.1 more states whereas BFS and DFS prove to be very inefficient. There is a significant increase in the average number of states explored while finding the path. For DFS its about 771 times more states and for BFS it is about 509 times more states.
Comparing average times taken by each of the algorithms, we notice that time differences are insignificant for small size problems, but difference is visible for times of 3x3 problems. BFS and DFS run with similar speed in both Solvable and unsolvable cases. A* algorithms takes significantly smaller times for solvable problem but it is terribly slower for unsolvable problems due to overhead of computing heuristic function for each node.

Algorithm	Solvable		Unsolvable	
	2x3	3x3	2x3	3x3
BFS	0.702	262.494	1.407	528.86
DFS	0.895	320.168	0.774	536.57
A*	0.949	45.433	1.694	1388.59

Fig. 3: Average Search time in Milliseconds

Comparison

A* vs BFS vs DFS Based on the results of these tests, we found that A* search against Breadth First Searching (BFS) and Depth First Search (DFS) algorithms and discovered that fewer states are being expanded with A*. A* expands paths that are already less expensive by using the heuristic and edge cost function. With problems that does not have a valid solution, the algorithms will traverse all of the possible state combinations and eventually return appropriate outcome. Since the problem example can result in multiple such cases, we decided to form two separate groups of such problem examples. We observed that A* outperforms other algorithms in cases where solution is reachable. On the other hand, the heuristic value calculation at every state and O(log(n)) complexity of priority queue makes execution significantly longer for cases where solution cannot be reached.

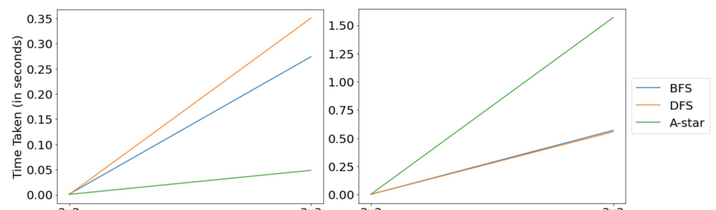

Fig. 4: Growth in time taken per between 2x3 and 3x3 search problems (solvable vs unsolvable)

Future Work

We observed that as the problem size grows, the number of possible states and the underlying graph of possible moves between these states. While size of puzzle problem does not seem as vast, the memory usage increases vastly over increase in size of puzzle board. 2x2 matrix has 12 unique states while 2z3 has 360 such states. This number grows to 181440 for a 3x3 puzzle. Since the path search algorithms implemented in our case keep track of huge list of visited and unvisited states in memory, the performance of the algorithm decreases very quickly. Poor Space Complexity is a major practical drawback in case of these algorithms.
Space optimized algorithms such as ID-DFS and IDA* can be explored for implemented for such cases, as these algorithms do not store the expanded states.

References

[1] Amey Darekar and Konstantin Tenman. *Benchmarking Path Search Algorithms*. 2022. URL: https://github.com/a3darekar/UT-Algo-Pathfinding-Benchmarks.

[2] Maciej Kurant, Athina Markopoulou, and Patrick Thiran. "On the bias of BFS (Breadth First Search)". In: *Proc. of the International Teletraffic Congress (ITC 22)* (Oct. 2010), pp. 1–8. DOI: 10.1109/ITC.2010.5608727.

[3] Xiang Liu and Daoxiong Gong. "A comparative study of A-star algorithms for search and rescue in perfect maze". In: (Apr. 2011). DOI: 10.1109/ICEICE.2011.5777723.

[4] Sheila Eka Putri, Tulus Tulus, and Normalina Napitupulu. "Implementation and Analysis of Depth-First Search (DFS) Algorithm for Finding The Longest Path". In: (Aug. 2011). DOI: 10.13140/2.1.2878.2721.