COT 4400

Analysis of Algorithms

Final Project

Members:

Maria Martinez

Miguelangel Rodriguez

Olivia Huegel

For this Final Project, we implement an 8- puzzle game using three different algorithms BFS, DFS and Dijkstra's algorithm. The purpose of this project is to find the cheapest solution for the 8- puzzle game. The 8- puzzle game has an initial state and the goal state with digits from 0 to 8, where 0 represents the empty place. The puzzle is a 3x3 board with 8 numbers, the goal state is ordering all the numbers on their correct position.

For the 8 puzzle game using BFS (Breadth-First Search), this algorithm is a traverse tree that visits all the nodes of each level at a time, from left to right. BFS implements a FIFO queue where new successors go at the end. The search begins with the first visit node construction then the path until it finds the goal state. BFS time complexity is O(V+E) where V is the vertex and E is the number of edges.This algorithm is slower than DFS and Dijkstra's algorithm. The way that this algorithm solves the 8 puzzle is slower and insufficient compared with the other searching algorithms that are faster and less complicated than this one.

The BFS algorithm uses three different functions move_up, move_down, move_right, and move_left to solve the 8 puzzle problem, the program moves until it finds that the current node is the expected node. The program prints recursive, it iterates from the parent node (the goal) to the initial path and then it prints the solution. The program contains three file codes, the main.cpp, Tile.hpp and Tilee.hpp file creates the node and the function to be implemented. The Node.cpp contains the function that prints and moves the nodes. The principal function to start moving is the blank_space function which iterates from around all the puzzles until it finds the 0, using the find() function then it finds the distance in order to find the position.

The move_up() function starts from the zero and then moves the node up. The move_down() starts from the zero and moves the node down. Move_right() and move_left()

functions are the same from the other function; they start with the zero and then move the node to the right or left by using the swap function, swapping the nodes. The main.cpp contains the initial trace and declares the final goal and it traces the solution, main.cpp calls the functions to move the node to the final goal. The main.cpp basically sets everything up and calls the function to find the goal traces.

Bfs algorithm is slow to perform the 8-puzzle game there is 9! Total possible configurations to solve the problem and only 9!/2 is the possible configuration to solve the problem this means that there are configurations that do not have a solution. Making this algorithm slow and not optimal.

Depth first search Algorithm is an incredible approach for traversing trees and graphs, it is a very accurate algorithm when it comes to ensuring the problem to be solved has an actual solution or path to follow. DFS differs with BFS or bread first algorithm when it comes to finding the shortest solution, BFS has a tremendous advantage due to its level to level visitation configuration. DFS instead travels alongs the vertices the way depth in the graph or tree to find the last node. Once it finds the last note it looks to the pointer to the parent of that child to start visiting the nodes until it finds a pointer to another child of that parent that has not been visited. The way in which it does it is by using a boolean to determine if that node was already visited. Moreover , to also keep track of direction, a stack is recommendable by using pop and push. In some applications queue and dequeue, those applications usually are BFS and A star algorithm. The optimized version of DMS implements recursion to solve the path in a recursive manner.

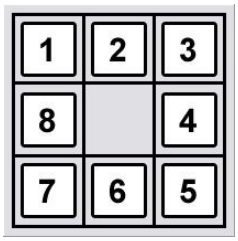
Depth first search Algorithm's implementation begins by initializing the parent state, getting the coordinates of the empty tile which is done by the coordinateFinder function, then

the next move is analyzed based on the possible move on the tile which could either be left, down, right or up. The max depth to traverse in the graph is set to five, the copy of the child is created to later switch with the new child. After that process, the new empty tile, is created, then if the child is equal to the goal state we end the execution, if not equal to the goal stated the program keeps looping by first popping from the list the node and recursively calling the function to repeat the process until the main if-condition is satisfied; Each iteration throws the new state of the puzzle with the tiles updated.

Dijkstra's algorithm is also known as the shortest path algorithm and is implemented in order to find the shortest path between nodes in a given graph or tree. For this project, we needed to implement Dijkstra's algorithm to solve a simple 8-puzzle problem and calculate the cost associated with moving the puzzle pieces. The cost of a move using Dijkstra's algorithm is equal to the number of the tile that is moved plus the number of tiles that are displaced. For example, the cost of moving tile #1 is $1 + the number of displaced tiles and the cost of moving tile #2 is $2 the number of displaced tiles and so on for the rest of the numbers in the puzzle. Because Dijkstra's algorithm finds the shortest path and solution to the puzzle, this will be the cheapest solution associated with getting the puzzle to the goal state. The program determines the cheapest solution and the number of tiles displaced while solving. This algorithm takes $O(|V| + |E| \log |V|)$ time and $O|V^2|$ by using an array where V is the number of nodes and E is the number of edges.

This program begins by reading in an initial or start state from a file and stores it into a 2D array. A state is initialized on each run and is solved using Dijkstra's algorithm that produces a solved puzzle like the one below. This algorithm uses a priority queue in order to

choose a tile with the lowest cost and calculates the distance of each of the other tiles in order to solve the puzzle. Dijkstra's algorithm will first create a list of all unvisited tiles and keep track of the unvisited tiles, the visited tiles, and the path between each of the tiles.



Summary of results

We found that the time complexities of the algorithms are as follows: Breadth-first search is O|E + V|, Depth-first search is O|E + V|, Dijkstra's algorithm is O(|V| + |E| log |V|).

Conclusions and findings

We implement these algorithms using different data structures in order to perform the search. BFS uses a Queue, DFS uses a Stack, and Dijkstra's algorithm uses a Priority Queue. We notice that BFS is a slow algorithm that takes too long to solve the puzzle BFS is fast in terms of finding a short solution but if it has to do to many movement to solve the puzzle it going to take too long to solve having configurations that does not have a solution and making the algorithm not optimal. BFS searches the nodes of each level at a time while DFS searches all the branches of a tree first, in our solution we realize that BFS gives us a faster solution than DFS. BFS gives us a solution in 5 paths while DFS 38 path makes it slower in this situation because it goes through all the branches of a tree first before checking if the node it is looking for is on the others level. BFS and Dijkstra's work in the same way but Dijltra's generates a faster solution making it a more optimal algorithm. We conclude that in cases of less

movements BFS is faster than DFS because the algorithm is faster, and due to Dijkstra's uses of Priority Queue it make this algorithm faster and more optimal than the other two algorithm.