Steven Bui, Ethan Ng, Alex Young
CS 325

Group Assignment 3

Description of Algorithm

To start, the function opens the input file and stores the integers in an array.

For our algorithm, we applied an approach similar to the breadth-first search. The first step is initializing the locations of the tokens. There are four separate variables that keep track of blue and red x and y coordinates. The blue token begins with coordinates (0, 0) and the red token begins with coordinates (n, n) where n = size - 1. These are stored together in a queue which represents the bag. There is a second queue which holds the number of steps and it works in accordance with the bag. There is an initialization of a dictionary called visited which will keep track of which nodes have already been visited. There is another dictionary called previsited that keeps track of count for tiles that the opposite color passed through, which forms half of the shortest path backwards.

Next, there is a while loop which continues to execute as long as the bag isn't emptied. A list of temporary coordinates will be popped from the bag and four new variables that keep track of blue and red x and y coordinates will be assigned to the list's variables. A temporary variable keeping track of steps will be assigned to the popped value of the steps queue. The visit and previsit dictionaries will mark these spots

There is a base case which checks to see if the top left and bottom right corners where the tokens start are both 0's. If this is the case, neither token will be able to move. This will break out of the while loop and the algorithm will return -1 by default to show that there is no solution.

If this is not the case, the next section has eight if statements (four for each token) to check all of the possible new positions on the grid. There are three conditions that determine whether or not a move is valid. First, a token cannot move beyond the bounds of the grid. Second, a token cannot move to a tile which is already occupied by another token. Third, the possible new position has not already been visited. If it meets all of these conditions, then the possible new position is appended to the bag and the temp variable for steps is incremented by one. This process continues until the win condition has been achieved which is determined if the blue token has reached the bottom right and the red token has reached the top left. If the algorithm has not reached the win condition and the bag has been emptied, it means that there was no solution and -1 is returned.

Optimization for this algorithm is far from perfect because our team had trouble getting faster code to run properly. In order to further improve the algorithm it would be best to use ‘redvisited’ and ‘bluevisited’ dictionaries separately in comparison to the other colors tile value. This would allow for far less duplicates, especially when looking at a table of 1s. In addition, by marking the tile values as visited at the start of the while loop, it increases the number of duplicates. If instead marking was done in the if statements and before the appending, it would stop tiles that are in the queue from being duplicated.

Runtime Analysis

$T(n) = O(n^2) * O(n^2) = O(n^4)$

The runtime for this algorithm ends up being O(n^4). The worst case scenario is a board full of ones. If we look at all the possible moves each piece can make, we find that each piece can move to each square in the board. This results in each piece having n^2 possible moves. Since we have 2 tokens, we have $n^2 * n^2$ possible moves, which gives us a worst case runtime of O(n^4).

Correctness of Algorithm

Base case:

- At the first phase n = 0, each corner of the puzzle is treated as a source. The sources are initialized and added to the queue. If the puzzle doesn't have any other tiles, then this is the shortest path

Induction hypothesis:

- If there is a set of moves (k) in the puzzle that go from beginning to end state, then there is a corresponding shortest path k that works.

Induction step:

- Show this by going through algorithm and corresponding each move to an edge in a graph
- If there is a set of moves less than k, the algorithm will find the path with fewer edges than k and replace it.
- The algorithm uses a breadth first search to search for every possible available path to the end
- By using this exhaustive search, the first found path will always be the shortest path because each edge that is being checked is in parallel count to the other moves in the same breadth