CS 51 Pset 6 Writeup
Spring 2023
Iris Lang and Cory Zimmerman

1 BenchmarkQueue

1.1 Results

max	list-based	stack-based	difference (list minus stack)
200	0.000633	0.000028	0.000605
2,000	0.040647	0.000137	0.040510
20,000	4.814887	0.002984	4.811903
50,000	50.340492	0.004614	50.335878

Table 1: BenchmarkQueue Stats in Seconds. Average over 5 Trials with Different Max Values.

1.2 Analysis

The stack-based queue outperforms the list-based queue in terms of efficiency on all max values that were tested. The difference in time (in seconds) is more significant as max increases, as can be seen in Table 1. Looking at the implementation of MakeQueueList and MakeQueueStack, we see that enqueuing an element for the list-based implementation requires linear time since the entire list is traversed every time an element is added. Dequeuing takes constant time.

For the stack-based implementation, we see that enqueuing and dequeuing altogether take constant time to complete. Dequeuing an element usually takes constant time unless a reversal is required, in which case it will take linear time in the number of enqueues. However, on average, enqueue and dequeue operations are amortized into constant time.

2 Tiles

2.1 Results

grid dimension	DFS	list BFS	stack (fast) BFS
2x2	0.041962	0.079870	0.078917
3x3	1515.040159	736.577988	730.088949
4x4	$>$5 mins	$>$5 mins	$>$5 mins

Table 2: Tiles Stats in Seconds. Average over 5 Trials with Different Grid Dimensions.

2.2 Analysis

DFS performs noticeably worse than BFS for the larger tiles puzzles. The list and stack BFS implementations have comparable run times.

Although both BFS and DFS have $O(V + E)$ time complexity, there are crucial differences between the two. The objective of the tile puzzle is to order the numbers 1 through N (here, we test N = 2,3,4) on the tile grid in a particular configuration, left to right, top to bottom. Note that the DFS algorithm continues making moves as long as states have not been visited until it is forced to backtrack. On the other hand, BFS tries all possible moves at each step. As such, one of these possible moves for BFS must provide the correct answer. In this way, BFS is more efficient, especially for larger grids, because for DFS, if the algorithm gets too deep into an incorrect solution, a significant amount of time will be spent backtracking. This happens repeatedly for every incorrect solution, incurring lots of penalty for incorrect solutions. BFS may take longer to progress from one step to the next due to the "breadth-first" nature, but it performs much better when lots of backtracking in DFS would be needed (in large tile grids).

3 Mazes

3.1 Results

grid dimension	DFS	list BFS	stack (fast) BFS
5x5	0.025034	0.090122	0.054121
10x10	0.085115	1.373053	0.909090
15x15	0.120163	18.100023	7.232904

Table 3: Maze Stats in Seconds. Average over 5 Trials with Different Grid Dimensions.

3.2 Analysis

DFS performs significantly better than BFS for the Maze puzzle. The stack BFS performs approximately twice as fast as the list BFS.

Our objective is to find a path from the top left corner of the grid to the bottom right corner of the grid. When comparing the graphics of the BFS and DFS solutions, we see that almost all squares are visited for BFS, whereas DFS optimizes much better and finds an efficient path to the target grid, close to a straight line. This makes sense if we recall what DFS and BFS are doing. BFS will traverse to all neighbors at every step of the puzzle as long as it is a legal move. Therefore, at every step of the puzzle, much more time is needed. However, by nature of this puzzle design, this is unnecessary. DFS takes a different approach, always moving closer to the target, making a new step as soon as it finds one possible neighbor to go to at every move. Perhaps some backtracking will be needed, but the backtracking distance will not be very large, so DFS is much faster for this puzzle.