Problem Set 6: Experiment Report

March 29, 2023

1 Introduction

In this problem set, we implemented a puzzle solving problem using OCaml higher order functors. A puzzle can be in any of a variety of states. The puzzle starts in a specially designated initial state, and we desire to reach a goal state by finding a sequence of move that, when executed starting in the initial state, reach the goal state. The goal state is to be reached by repeated moves of this sort. Solving goal-directed problems of this sort requires a search among all the possible move sequences for one that achieves the goal.

The speed or the effectiveness of the searc process towards the goal state from the initial state depends on the order in which states are taken from the collection of pending states as the search proceeds. If the states taken from the collection are those most recently added to the collection (last-in, first-out, that is, as a stack), the tree is being explored in a depth-first manner. If the states taken from the collection are those least recently added (first-in, first-out, as a queue), the exploration is breadth-first.

2 Implementation

The functor implementations of a stack (Last-In-First-Out) and a queue (First-In-First-Out) were already provided in `collections.ml`. In addition, we implemented a `MakeQueueStack` functor for generating a queue collection, where the queue is implemented using two stacks. We used all three functors in the puzzle experiments.

The `PUZZLEDESCRIPTION` signature provides the specifications for different puzzles, specifically information about the states and moves, initial and goal states, the neighbors structure of the puzzle, etc. In this problem set, we worked with two specific forms of puzzles, namely tiles and mazes, whose specifications were provided by `tiles.ml` and `mazes.ml`. The specifications for tile and maze puzzle conform to the signature provided by `PUZZLEDESCRIPTION`.

The heart of the puzzle implementation is provided by the `solve()` function provided by the higher order functor `MakePuzzleSolver`. This higher order functor enables us to generate different puzzle solvers, specifically one for tiles and another for mazes in this exercise. The `solve()` function uses two data structures, one for maintaining the pending states and another for maintaining the visited states, which have been removed from the pending states. The search proceeds by taking a state from the pending collection (call it the current state). If the current state has already been visited (that is, it?s in the visited set), we can move on to the next pending state. But if it has never been visited, we add it to the visited set and examine it further. If it is a goal state, the search is over and appropriate information can be returned from the search. If not, the neighbors of the current state are generated and added to the pending collection, and the search continues.

The higher order functor `MakePuzzleSolver` allows us to generate different collection regimes - stacks (`MakeStackList`), queues (`MakeQueueList`, `MakeQueueStack`), etc. - which leads to different search regimes - depth- first, breadth-first, etc. The argument functor can be used to provide a collection of elements of any type.

3 Experiments

We conducted a number of experiments with both the Tile puzzles and Maze puzzles to evaluate the search performance with depth-first search (DFS), breadth-first search (BFS), and a faster breadth-first search (Fast BFS), whose implementations were provided in `collections.ml` in the form of `MakeStackList`, `MakeQueueList` and `MakeQueueStack` respectively.

3.1 Experiments with Tiles

Using the puzzle frame work that we implemented for a generic collection, we conducted experiments for four different puzzle sizes. They are namely,

1. 2×2 Tile
2. 3×3 Tile
3. 4×4 Tile
4. An arbitrary unsolvable Tile.

For each of the above cases (except for the last case), we measured the run-times for each of the following:

1. Depth-first search (DFS),
2. Breadth-first search (BFS), and
3. Fast breadth-first search (Fast BFS)

The experiments were constructed using the initial framework provided in `tests.ml`. Following is a sample run on both a 2×2 Tile puzzle and a 3×3 Tile puzzle:

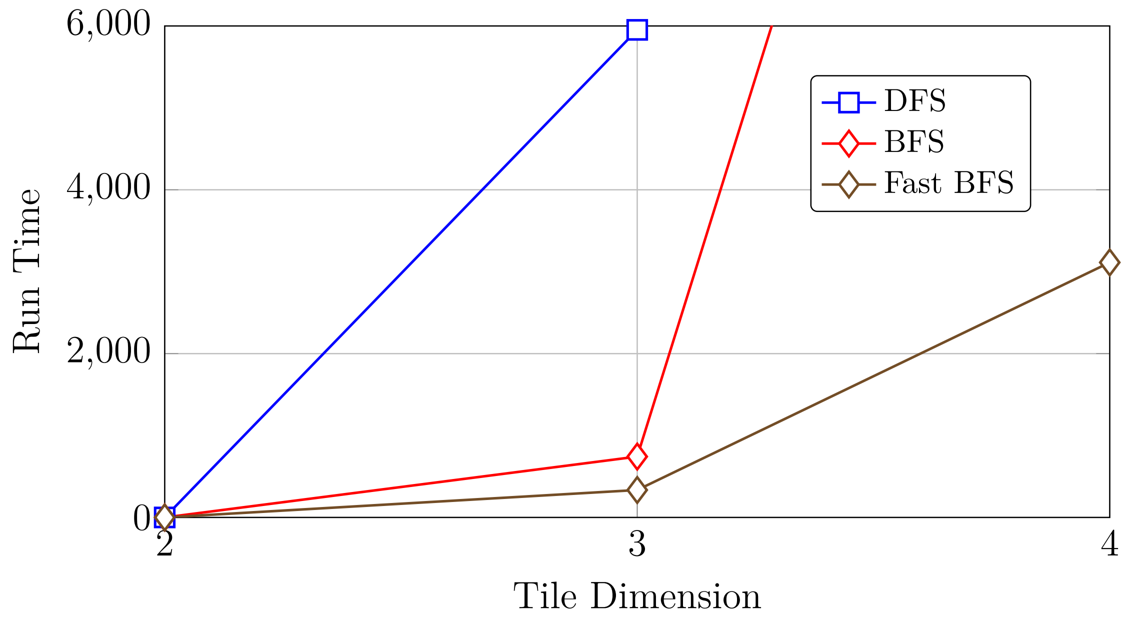
```
TESTING RANDOMLY GENERATING 2x2 TILEPUZZLE...
2x2 Regular DFS time:
time (msecs): 0.070095
2x2 Regular BFS time:
time (msecs): 0.138044
2x2 Faster BFS time:
time (msecs): 0.140190
DONE TESTING 2x2 TILE PUZZLE

TESTING RANDOMLY GENERATING 3x3 TILEPUZZLE...
3x3 Regular DFS time:
time (msecs): 5952.724934
3x3 Regular BFS time:
time (msecs): 743.596077
3x3 Faster BFS time:
time (msecs): 335.777998
DONE TESTING 3x3 TILE PUZZLE
```

As we can see from the above experimental results, the depth-first search using a stack is the fastest among the three choices for the simplest 2×2 Tile puzzle, while the Fast Breath-First search using the two stack implementation of the queue is the worst. However, the search efficiency is exactly opposite for the 3×3 Tile puzzle, where the Fast Breath-First search (using the `MakeQueueStack` functor) is more than twice as fast as the regular breadth-first search (using the `MakeQueueList` functor) and is a blazing 17 times faster than the depth-first search (using the `MakeStackList`. Following table shows the run times of various tile puzzles in each case:

Tile Size	DFS (ms)	BFS (ms)	Fast BFS (ms)
2×2	0.070095	0.138044	0.140190
3×3	5952.72	743.60	335.78
4×4	-	19217.28	3113.72

We note that in the case of a 4×4 Tile, the depth-first-search implementation did not finish running even after 30 minutes, which prompted us to abort the run. Following is a plot of tile dimension versus run-time to solve the puzzle:



The above plot leads us to conclude that as the dimension of the tile increases, the Fast-Breath-First search implementation using the `MakeQueueStack` functor) will scale better than the other two implementations. Even though we had added the

`CantReachGoal` exception check in the code, none of the three implementation ever hit this exception when the dimension was set to four. The reason for that is that as the dimension is increased, the search space exponentially increases, which would require significantly longer time to exhaust all the possibilities before raising the exception.

3.2 Experiments with Mazes

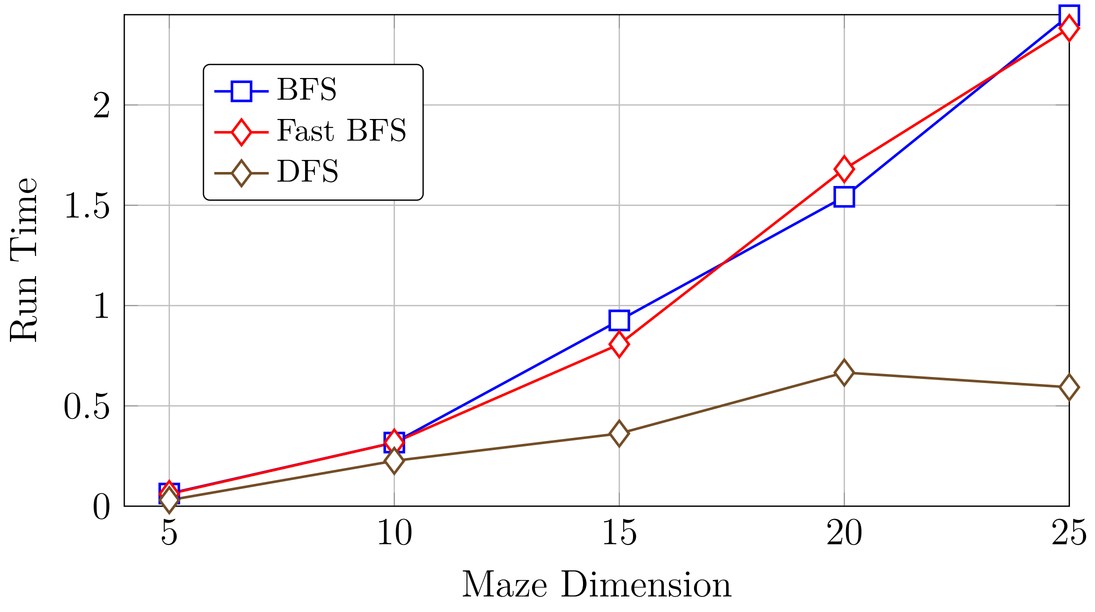
Mazes are two dimensional grids where the elements represent open space or walls. In a maze, the state would be represented as the current position in the maze, and the moves would allow one to change the position by walking up, down, left, or right. The neighbors of a state would be the positions that one could move to in one step using those moves, keeping in mind that not all four move types are possible from a given state.

We started with a 5×5 maze, and gradually increased the sizes to 10×10, 15×15, 20×20, and 25×25. For each of the above cases (except for the last case), we measured the run-times for each of the following:

1. Depth-first search (DFS),
2. Breadth-first search (BFS), and
3. Fast breadth-first search (Fast BFS)

For each of the four maze dimensions above, we obtained the run time to reach the goal position, which was the top-right corner of the grid from the initial position, which was the bottom-left corner of the grid. We found that consistently the Depth-First-Search implementation (using the `MakeStackList` functor) outperformed the other two implementations. The results are listed in the table below and are graphed in the plot below.

Maze Size	DFS (ms)	BFS (ms)	Fast BFS (ms)
5×5	0.030041	0.062943	0.061035
10×10	0.226021	0.062943	0.317097
15×15	0.362158	0.926018	0.807047
20×20	0.666142	1.542091	1.680136
25×25	0.592947	2.447128	2.382994

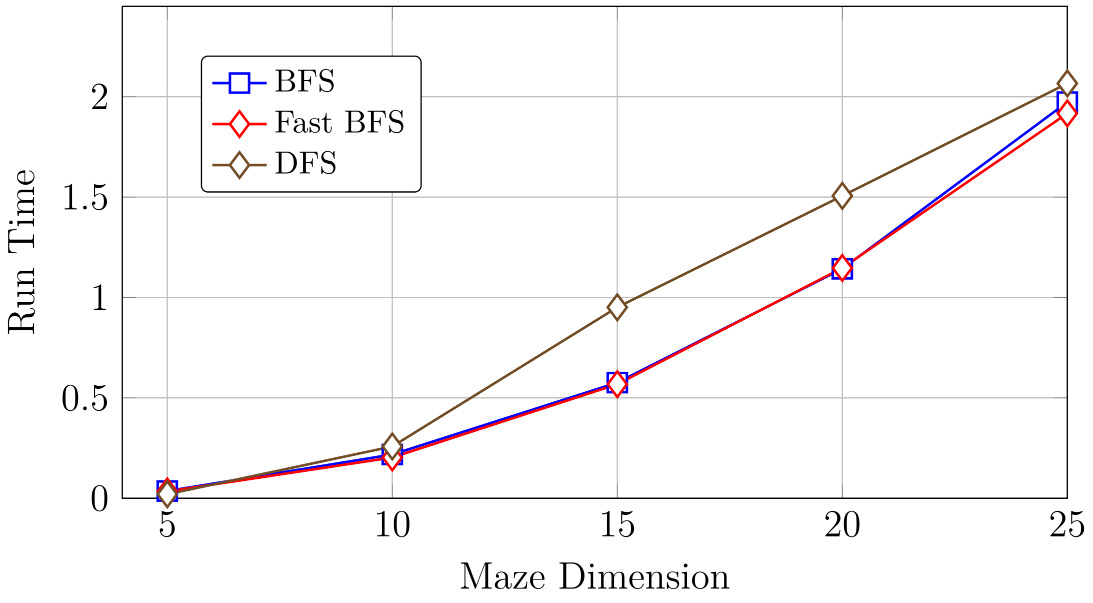


The above plot shows that when the square maze has its initial and goal positions are diagonally opposite to each other, the Depth-First-Search implementation (using the `MakeStackList` functor) is significantly faster than the other two implementations.

When we changed the goal position to the left-top corner, we observed that both the Breath-First-Search implementation (using the `MakeQueueList` functor) and the

Fast Breath-First-Search implementation (using the `MakeStackQueue` functor) performed somewhat better than the Depth-First-Search implementation (using the `MakeStackList` functor). The results are listed in the table below and are graphed in the plot below.

Maze Size	DFS (ms)	BFS (ms)	Fast BFS (ms)
5×5	0.020027	0.036001	0.033855
10×10	0.257969	0.217915	0.203133
15×15	0.951052	0.576019	0.567913
20×20	1.506090	1.142979	1.147032
25×25	2.066135	1.973152	1.916885



4 Conclusions

We investigated three different search implementations in this problem set, namely, Breadth-First-Search (using the `MakeQueueList` functor) , Fast-Breadth-First-Search (using the `MakeQueueStack` functor) and Depth-First-Search (using the `MakeStackList` functor). Our experiments showed that the Fast-Breadth-First-Search (using the `MakeQueueStack` functor) implementation is the fastest for tile puzzles, specially when the dimension of the tile increases above 3×3. The run time for the other two strategies were significantly larger than that of the Fast-Breadth-First-Search (using the `MakeQueueStack` functor) implementation for solving tile puzzles. This is because, on the average for large n, the implementation of `MakeQueueStack` takes constant time for an "add" operation, where as the `MakeQueueList` takes $O(n)$ time to add the element as it needs to traverse the entire list. The `MakeStackList` implementation takes a constant time to add an element, however the search operation takes is significantly longer as the goal state is invariably not on the top of the stack. Hence the choice of the Fast-Breadth-First Search implementation (using the `MakeQueueStack` functor) is the best choice for implementing a tile puzzle presented in this problem set.

We observed that the run time to solve the puzzle was dependent on where the goal state was located. Specifically, when the square maze has its initial and goal positions are diagonally opposite to each other, the Depth-First-Search implementation (using the `MakeStackList` functor) is significantly faster than the other two implementations. On the other hand, when we changed the goal position to the left-top corner, we observed that both the Breath-First-Search implementation (using the `MakeQueueList` functor) and the Fast Breath-First-Search implementation (using the `MakeStackQueue` functor) performed somewhat better than the Depth-First-Search implementation (using the `MakeStackList` functor). In other words, when the goal state was farther from the initial state, the Depth-First-Search approach solved the puzzle faster than the other two. This is because the Depth-First-Search implementation is more amenable for searching deep within a search tree before switching to another tree.

In conclusion, this problem set allowed us to compare the design choices for specific searching strategies such as Depth-First-Search, Breadth-First-Search or a modified Fast-Breadth-First-Search that could be employed in solving a puzzle. We also learned how to use various collections for improving the searching strategies. Collections are a generalization over a variety of data structures including stacks, queues, and priority queues. Specifically, we learned how to use higher order functors to develop generic modules that can be used easily across different collections such as stacks, priority queues, queues implemented using double stacks and so on, without having to worry about specific design details pertinent to a given data structure.