Use of BFS and DFS Algorithms for Solving the Rescue Plan Puzzle Toy

Sebastian Hung Yansen - 13523070
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: sebastianhung25@gmail.com , 13523070@std.stei.itb.ac.id

***Abstract*—This paper explores the use of search algorithms, namely BFS and DFS, for solving the Rescue Plan puzzle toy through graph traversal. BFS searches neighboring nodes within a graph whilst DFS searches a specific path as deep as possible. By modelling the puzzle as a state-space graph, both algorithms are implemented, tested, and compared with each other. The results show that both methods are equally effective but their performance varies based on the puzzle's complexity. BFS guarantees the shortest path to the solution and excels on shallow puzzles though suffers when met with a deep and branching graph. DFS does not guarantee the shortest path but it can be more efficient in terms of memory and runtime especially for deeper graphs.**

Keywords—Graph Traversal; BFS; DFS; State-Space Search

I. Introduction

Graph traversal has been used in many cases of problem solving. Some of them are simple while others are more complex. Graph traversal, however, is commonly used for pathfinding. Even problems, that from a glance don't look like it can be solved using graph traversals, can actually be solved using pathfinding. These problems exist within our daily lives and we always meet them every day including the things we own like toys.

Toys appeal to people of many ages including children. Toys are intrinsically motivating, says Abrams and Kaufmann [4]. As playing with toys can set the foundation for reading, writing, and mathematical reasoning, a lot of toys are focused on developing problem-solving skills in children from an early age. In the National Association for the Education of Young Children journal, an article was written by Segatti, Du-Paul, and Keyes who state that children innovate with toys and these innovations are signs that children are learning to use their thinking skills to solve problems [4]. Therefore, many toys that are focused on problem-solving are made even to this day. These toys have certain attributes that benefits those who play with it.

Some of these toys, although made for children, can be solved through algorithms that make use of graph traversal. One of these toys is called Rescue Plan. The puzzle in the toy is setup where there are lifeboats and sinking passengers at sea. The player moves the lifeboats to rescue all the passengers from the water. The puzzle is solved when all the sinking passengers are seated safely in a lifeboat.

Fig. 1.1. Rescue Plan Puzzle Toy

The challenge comes from the size of the board as well as how free the lifeboats can go. On top of that, the player must think about what order the passengers must climb on to the lifeboat as to not anchor the lifeboat and block the other lifeboats from saving other passengers as well as which lifeboat should pick up which passengers. On harder difficulties, it becomes much harder with the number of passengers and lifeboats in play.

With the amount of freedom the lifeboats can move, it might seem daunting to map out the problem. However, it is possible to solve this problem with graph traversal using a state space graph that represents the states of the board itself. This paper will discuss and explore how BFS and DFS algorithms can be used to solve the Rescue Plan puzzle. This paper will also analyze and compare both applications of the algorithms, demonstrating the performance trade-offs between the two algorithms through testing and experimentation.

II. Theoretical Basis

A. Graphs

According to Das, R., & Soylu, M., Graphs themselves serve as powerful mathematical representations for modelling relationships, networks, as well as structures within diverse domains. A variety of algorithms have been made with each of

them designed to address specific graph-related problems. A graph represents data in an organized way through a diagram. The diagram shows the relationship between nodes using Vertices as nodes and Edges as lines that connect between nodes. A single graph G is defined as G = (V, E) as shown below

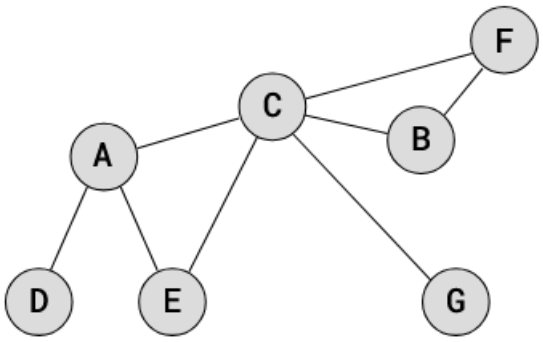

Fig. 2.1. Graph representation using Vertices and Edges

Source:
https://www.w3schools.com/dsa/dsa_theory_graphs.php

These vertices and edges can represent different things from intersections on a road all the way to computer networks. Another thing a graph can represent are states of a certain thing, called a state space graph. A state space graph is a graph that is constructed with vertices representing states and edges representing transitions between each state that connect from a state to its successors. These states are only represented once within a single graph.

B. Graph Traversal

Graph traversal is the process of exploring vertex values in a graph data structure. Most search algorithms are used to solve graph problems by examining every vertex and every edge.

Search algorithms themselves have a few properties to keep in mind when comparing with one another. These properties include completeness, optimality, time complexity, and space complexity. Completeness addresses whether an algorithm guarantees it will find a solution if at least one exists. For a finite graph with no cycles, most traversal algorithms are complete. Optimality represents if a solution is guaranteed to be the best solution out of all, typically the one with the lowest path cost. Time complexity is a measure of time for an algorithm to complete its task. Space complexity is the maximum storage space required during the search; it also more or less represents the complexity of the problem. This is often a critical factor in large state-space graphs where storing visited nodes or the frontier of nodes to be explored can become very demanding.

When an algorithm traverses a graph, it effectively unrolls the graph's connections to form a search tree. The root of this tree is the initial state, its branches represent the actions or moves, and the nodes represent the states reached during the search process. This search tree is a representation of the actual paths explored by an algorithm, which can be a subset of the entire state-space graph. The strategy an algorithm uses to build and explore this tree is what differentiates one from another.

Mainly there are two types of graph traversal techniques or search algorithms: Breadth First Search and Depth First Search.

C. BFS

Breadth First Search or BFS is a graph traversal algorithm that explores all neighboring nodes at the present depth level before moving on to the other nodes on the next depth level. Assuming G(V, E) is a graph, applying breadth first search is as follows:

1. Designate a starting vertex (v)
2. Visit all neighboring vertices from v that are connected with an edge
3. Visit the other neighboring vertices that haven't been visited and continue the process until a specific vertex is found or all vertices have been visited.

D. DFS

Depth First Search of DFS is another graph traversal algorithm that explores a specific path or branch as far as possible before backtracking to the previous vertex. Assuming G(V, E) is a graph, applying depth first search is as follows:

1. Designate a starting vertex (v)
2. Pick a path connected to v and traverse it
3. Continue to traverse it until a specific vertex is found or until a dead end is met
4. If traversing a path meets a dead end, backtrack and go down an unexplored path
5. This process repeats until a specific vertex is found or all vertices have been visited.

E. Comparison Between BFS and DFS

Aside from how each algorithm explores a graph in their own ways, there are other comparisons that can be seen between them. From the perspective of data structures, DFS utilizes a stack. A stack is a special kind of list in which all insertions and deletions take place at one end, called the '*top*' [1]. Another name for a stack is a "LIFO" or "last-in first-out" list. On the other hand, BFS utilizes a queue, another special kind of list where items are inserted at the end (the *rear*) and deleted at the other end (the *front*) [1]. A queue has a different name called "FIFO" or "first-in first-out".

Aside from its' data structure, both BFS and DFS have other differences. Because BFS uses a queue system, large graphs can be very demanding as it has to store all the nodes at a certain level of depth in the queue. DFS uses a stack and doesn't need to store every vertex like in BFS and consumes less memory than BFS. However, because DFS searches a specific path as deep as possible, there's a chance that it won't find the shortest path within a graph. BFS is more fitting for finding the shortest path in a graph. Finally, the space complexity for both BFS and DFS are different. Both have a space complexity of O(V) but V in DFS represents the maximum depth whilst V in BFS represents the maximum

number of nodes at a single level in the search. This distinction is critical as BFS may consume more memory in a wide and shallow graph whilst in a narrow and deep graph, DFS remain memory-efficient. Aside from those differences, BFS and DFS is similar in completeness as both algorithms will search the entire graph and return a solution if it exists as well as time complexity of O(V + E) where V represents the number of nodes whilst E represents the number of edges.

Both BFS and DFS also falls in the category of Uninformed search or blind search. What it means is that both algorithms explore a graph without any specific knowledge about the entire graph nor the goal or path to reach the goal. They only have information on how to traverse the graph and identify nodes and goal nodes.

III. Implementation

First and foremost, the player positions the lifeboats and the passengers however they like or according to the challenge cards that the toy provides. Once the lifeboats and passengers are in position, the player can start playing and saving the sinking passengers. Each boat may move forwards, backwards, or drift from side-to-side and cannot turn nor move diagonally. When a sinking passenger is close enough to a lifeboat, they can and will climb aboard the lifeboat. The passengers may not switch positions and when a lifeboat is full, it must remain anchored to the spot and cannot move anymore.

We will be defining and mapping a few theories towards the puzzle itself. First and foremost, we will be using states. Each state saves and encapsulates the entire board and represents the current position of the lifeboats as well as the current position of the remaining sinking passengers. The positions themselves are represented using x and y coordinates on the board. We will be using a few of the challenge cards that shows the position of each passenger and lifeboats for its initial state. Its' final state is met when every passenger has successfully been picked up by the lifeboats. The transitions between each state (or in the case of graphs, the edges) represent valid moves. The valid moves that are allowed follows the games' rules where the lifeboats are allowed to move up, down, left, and right as long as it's not blocked by a passenger or another lifeboat.

For its' implementation, we will be using the java programming language that models the puzzle as a state space graph and uses the search algorithms mentioned in Chapter II. We have classes for each object which consists of lifeboats, passengers, coordinates, and the state. We also have classes for solving, moving the lifeboats, and the main class. The state class stores the lifeboats and the passengers using a List. On top of that, the state class stores hashes for performance and to keep track of visited states.

For reading the initial state of the board, the program will read from a txt file which consists of dots and letters to represent each lifeboat and passenger. An example can be seen below

TestCase.txt

```
.P...P
B<A..P
B..P^.
VP..C.
```

The search process begins with processing the initial state from the provided txt file. The program then solves the initial state using the selected algorithm, generating successors and exploring them until a goal state is found. Both algorithms use a set to keep track of the visited states as to prevent infinite loops in the graph traversal itself. The goal state is reached when the number of sinking passengers on the boat reaches zero.

Each vertex or node stores a state and the reference to its parent. Once the goal state has been found, a method is called to walk back from the goal state all the way to the initial state to reconstruct the path and represent the solution path itself.

A. BFS Implementation

The BFS implementation uses a queue as it explores all neighbors at the current depth before moving to the next level. the pseudocode is detailed in Fig 3.1.

```
function solveBFS(initialState)
  let frontier be a new Queue
  let explored be a new Set
  let startNode be a new Node containing initialState
  add startNode to frontier
  add initialState to explored
  while frontier is not empty
      let currentNode <- remove front item from frontier
      if currentNode's state is the goal
          return the path from currentNode
      end if
      for each successorState from currentNode's state
          if explored does not contain successorState
              add successorState to explored
              let newNode be a new Node containing successorState and currentNode
              add newNode to frontier
          end if
      end for
  end while
  return No Solution Found
end function
```

Fig. 3.1. BFS Logic Implementation

B. DFS Implementation

The DFS implementation uses a stack as it explores a single path as deeply as possible before backtracking. The pseudocode is detailed in Fig 3.2.

```
function solveDFS(initialState)
   let frontier be a new Stack
   let explored be a new Set
   let startNode be a new Node containing initialState
   push startNode onto frontier
   while frontier is not empty
      let currentNode <- pop item from frontier
      if explored contains currentNode's state
            continue to next loop iteration
      end if
      add currentNode's state to explored
      if currentNode's state is the goal
            return the path from currentNode
      end if
      let successors be the list of next states from currentNode's state
      reverse the order of successors
      for each successorState in successors
            if explored does not contain successorState
                  let newNode be a new Node containing successorState and currentNode
                  push newNode onto frontier
            end if
      end for
   end while
    return No Solution Found
end function
```

Fig. 3.2. DFS Logic Implementation

IV. Experiment

The initial state we will be using is taken from the txt file from the previous chapter with 3 lifeboats and 5 sinking passengers. We will be testing it with both DFS as well as BFS and we will compare the runtime and the number of nodes (in this case, steps) to reach the goal state.

```
Initial State:
. P . . . P
B < A . . P
B . . P ^ .
V P . . C .
. . . . C .
. . . . . .
Boat A: 0/2 passengers
Boat B: 0/1 passengers
Boat C: 0/2 passengers

Attempting to solve using BFS search...

Solution found in 16 moves!
Total runtime: 1204 ms
```

Fig. 4.1. BFS TestCase1 Results

```
Initial State:
. P . . . P
B < A . . P
B . . P ^ .
V P . . C .
. . . . C .
. . . . . .
Boat A: 0/2 passengers
Boat B: 0/1 passengers
Boat C: 0/2 passengers

Attempting to solve using DFS search...

Solution found in 24 moves!
Total runtime: 1236 ms
```

Fig. 4.2. DFS TestCase2 Results

As shown, DFS was marginally slower than BFS by 32 milliseconds. While small, it shows that both BFS and DFS have a clear difference. The shorter runtime indicates the optimal path to the solution is shallow and is not too deep. Not

only that, the number of steps shown in BFS is smaller than the one shown in DFS.

Does this mean that BFS automatically always provide the best path with the shortest time? Not necessarily. Take for instance, this the following initial state

TestCase2.txt

```
^.P.P.
A..P..
A..<BB
...P..
```

At a glance, it looks much more simpler than the one before. But when we try to solve it with both DFS BFS, the reuslts are almost different with the testcase from before.

```
Initial State:
^ . P . P .
A . . P . .
A . . < B B
. . . P . .
. . . . . .
. . . . . .
Boat A: 0/2 passengers
Boat B: 0/2 passengers

Attempting to solve using BFS search...

Solution found in 29 moves!
Total runtime: 382 ms
```

Fig. 4.3. BFS TestCase2 Results

```
Initial State:
^ . P . P .
A . . P . .
A . . < B B
. . . P . .
. . . . . .
. . . . . .
Boat A: 0/2 passengers
Boat B: 0/2 passengers

Attempting to solve using DFS search...

Solution found in 229 moves!
Total runtime: 231 ms
```

Fig. 4.4. DFS TestCase2 Results

It turns out that the runtime for DFS is shorter than BFS. This is not a mistake as it can happen depending on how the graph is shaped. If we take a look at Fig 4.4, it took DFS 229 moves or steps to find the goal state whilst BFS only took 29.

According to a comparison of traversal strategies between DFS and BFS, DFS is more effective in memory-constrained settings and deep searches whilst BFS is better at discovering the shortest paths and providing comprehensive coverage [8]. The faster runtime for DFS, despite finding a much longer path, indicates that BFS was slowed down by the number of states it had to store and traverse at each level. DFS was able to explore a single path to a solution more quickly, avoiding the combinatorial explosion that hampered BFS. These results can be further improved by using variations of DFS and BFS or by using a heuristic approach. This way, it is possible to 'indirectly' lead the algorithms towards the goal state and therefore the algorithm won't need to check pointless states.

V. Conclusion

Algorithm strategies provide general approaches in solving problems algorithmically so that it can be applied to various problems. Based on the analysis, implementation, as well as the experimentation that has been done, it can be concluded that both DFS and BFS are reliable and effective searching algorithms within a graph. Their performance is highly dependent on the puzzle's structure. The experiment demonstrates that BFS is superior in finding the shortest and most optimal path to the solution and is most efficient on puzzles with shallow solution depths. Conversely, DFS can outperform BFS in terms of runtime on puzzles with high branching factors, proving more effective when memory efficiency is critical even if it returns a non-optimal path. As such, there is no single 'best' algorithm as each algorithm has its own trade-offs between time complexity, space complexity, as well as the solution optimality that people must consider.

There are many more search algorithms out there that can be used other than BFS and DFS, even better ones. Even so, both BFS and DFS are still fundamental to the study of algorithms for their simplicity and clear trade-offs making them essential tools for problem solving. A clear direction for future work would be to implement and compare different algorithms such as A*, UCS, and even Greedy Best First Search.

Acknowledgment

The writer of this paper would like to thank the Lord for His guidance and mercy for giving me the ability and strength to finish this paper. The writer would also like to thank their parents, friends, and lecturer for their support in knowledge and material. The writer is grateful to everyone that they've met and those who've helped the writer go through tough parts of their life.

REFERENCES

[1] J. E. Hopcroft, J. D. Ullman, and A. V. Aho, *Data structures and Algorithms*. Boston, MA, USA: Addison-Wesley, 1983.

[2] J. Iyanda. "A Comparative Analysis of Breadth First Search (BFS) and Depth First Search (DFS) Algorithms."

[3] K. J. Hamad, and S. S. Mahmood. "An Analytical Study of Graph Algorithms: An Overview of Graph Theory." *QALAAI ZANIST SCIENTIFIC JOURNAL,* vol. 10, no. 1, pp. 1217–1238, 2025.

[4] J. C. V. Clavio and A. C. Fajardo. "Toys as instructional tools in developing problem-solving skills in children." *Education Quarterly,* vol. 66, no. 1, pp. 1–15, 2008.

[5] R. Das and M. Soylu, "A key review on graph data science: The power of graphs in scientific studies," *Chemometrics and Intelligent Laboratory Systems*, vol. 240, p. 104896, 2023.

[6] N. Banerjee, S. Chakraborty, V. Raman, S. R. Satti "Space efficient linear time algorithms for BFS, DFS and applications.", *Theory of Computing Systems* vol. 62, 2018, pp. 1736–1762

[7] T. Evritt and M. Hutter, "Analytical results on the BFS vs. DFS algorithm selection problem: Part II: Graph search." *AI 2015: Advances in Artificial Intelligence: 28th Australasian Joint Conference, Canberra, ACT, Australia, November 30--December 4, 2015, Proceedings 28*. Springer International Publishing, 2015.

[8] A. Z. Ismaeel and I. M. I. Zebari, "Comparing Traversal Strategies: Depth-first Search vs. Breadth-first Search in Complex Networks." *Asian Journal of Research in Computer Science* vol. 18, no. 2, pp. 60–73, 2025.

[9] K. Khandelwal, R. Gupta, "Search-Notes1," Dept. of Computer Science & Engineering, Univ. of Washington, Seattle, WA, USA, Lecture Notes, Autumn 2023. [Online]. Available: https://courses.cs.washington.edu/courses/cse473/23au/notes/Search-Notes1.pdf. [Accessed: Jun. 22, 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juni 2025

Sebastian Hung Yansen / 13523070