University of St Andrews

Assignment 2

Search

Student ID: 120022067

University of St Andrews

CS5011 Artificial Intelligence Practice

Contents

List of Figures

List of Tables

Listings

1 Introduction

Rescue and evacuation are essential operations in the even of natural disaster. In many cases, the evacuation route can be blocked by various obstacles and collapses. Autonomous robots can be used in these situations by searching disaster areas for victims, to avoid risking the lives of human rescue teams (Toniolo 2017*a*).

The aim of this assignment is to implement and evaluate a set of AI search algorithms that allow the an agent (robot) to navigates a building badly affected by an earthquake, locates a victim, and transport him/her to a safe location on the map.

1.1 Parts Implemented

Part 1: Implemented every feature stated in the requirements. The algorithms implemented in part 1 are Breadth First Search and Depth First Search. Full details of the implementation can be found in Section 3.

Part 2: For part 2, Best First Search and A* Search are implemented. The chosen heuristics are Manhattan distance and Euclidean distance. An in-depth discussion of part 2's implementation can be found in Section 4

Part 3: Implemented a solution for the situation where Bob, cat, and dog are at position marked 'B'. The chosen algorithm for finding path to 'B' and 'G' is A* (heuristic choices are the same as part 2). An in-depth discussion of part 3's implementation can be found in Section 5

2 Literature Review

2.1 Breadth First Search

Breadth First Search (BFS) is a simple search algorithm in which the root node is explored first (Russell & Norvig 2009). As shown in Figure 1, all successor nodes in the same depth are expanded before any nodes at the next level are expanded. Russell & Norvig (2009) explains that node expansion in BFS can be done by using a FIFO queue for a frontier, which means that successor nodes go to the back of the queue. This allows the old nodes to be expanded first.

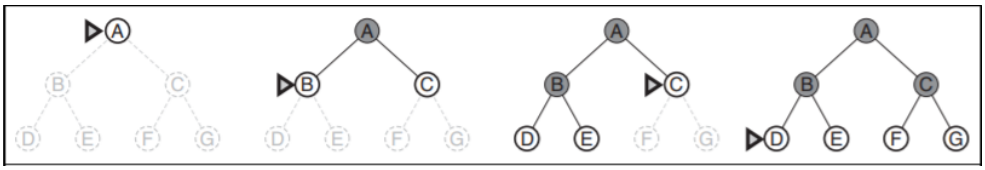

Figure 1: Node expansions in Breadth First Search (Russell & Norvig 2009).

2.1.1 Analysis

According to Russell & Norvig (2009), BFS can be considered complete because BFS is able to find the goal if it is at some finite depth d (the branching factor b should also be finite). Russell & Norvig (2009) states that BFS is optimal if the path cost is a nondecreasing function of the depth of the node. An example of such scenario is that all actions have the same cost (Russell & Norvig 2009).

According to Russell & Norvig (2009), the time complexity of BFS is $O(b^{d+1})$, where b is the branching factor (number of children at each node) and d is the depth of the goal node. The space complexity of BFS is $O(b^d)$ (Russell & Norvig 2009). With the exponential time and space complexities, it is clear that BFS is not very efficient. It requires massive amount of time and space if the goal node is located in lower depth. Figure 2 shows time and memory requirements for BFS with branching factor $b = 10$ assuming that 1 million nodes can be generated per second and that a node requires 1000 bytes of storage (Russell & Norvig 2009).

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 2: Time and memory requirements for Breadth First Search (Russell & Norvig 2009).

2.1.2 Applications of Breadth First Search

Some applications of BFS are listed below (Jain 2017).

1. **Peer to Peer Networks:** BFS is used to find all neighbor nodes in Peer to Peer Network like BitTorrent.
2. **GPS Navigation systems:** BFS can be used to find neighboring locations.
3. **Broadcasting in Network:** broadcasted packet can follow BFS to reach all nodes.
4. **Social Networking Websites:** BFS can be used to find people within a given distance 'k' from a person.

2.2 Depth First Search

Russell & Norvig (2009) explains that Depth First Search (DFS) always expands the nodes until it reaches the deepest level of the search tree (no more successor nodes). The progress

of the search is shown in Figure 3. As those nodes are expanded, they are removed from the frontier, so then the search return to the next deepest node that still has unexplored successors (Russell & Norvig 2009). DFS uses a LIFO queue for a frontier, which means that successor nodes go to the front of the queue. This allows the new nodes to be expanded first (Russell & Norvig 2009).

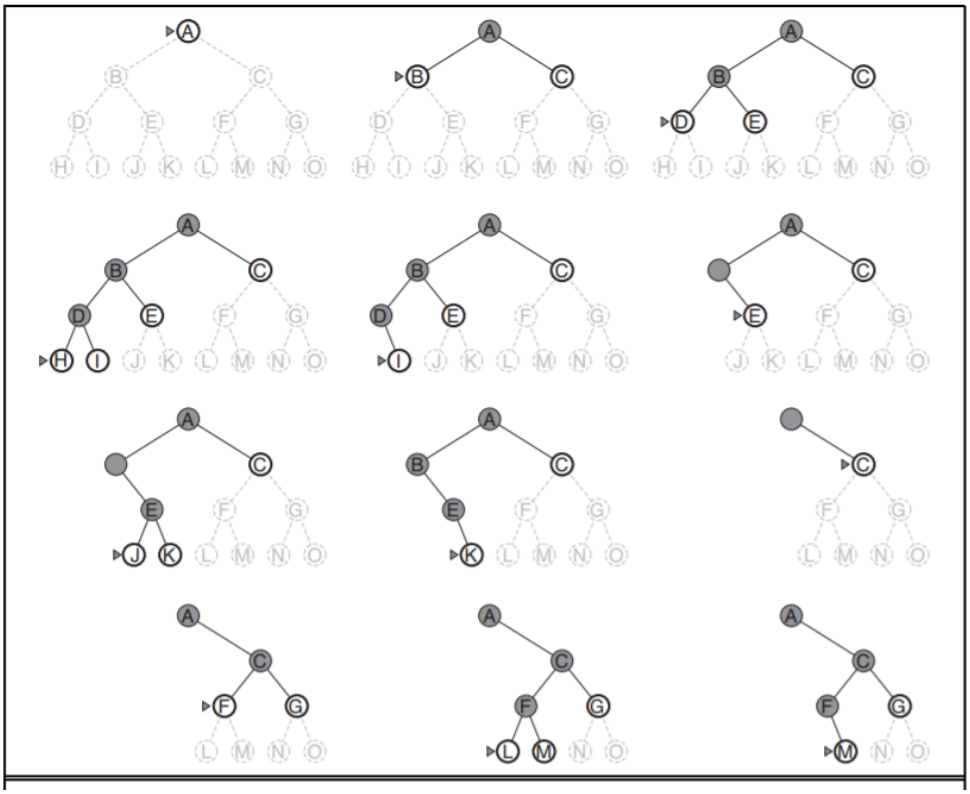

Figure 3: Depth First Search on a binary tree. (Russell & Norvig 2009).

2.2.1 Analysis

DFS is not complete because it can get stuck in an infinite non-goal path forever (Russell & Norvig 2009). However, it can be considered complete in this assignment because the maps are 10x10 grid, which means that the state space is finite. DFS will eventually explore every position on the maps. Russell & Norvig (2009) also states that DFS is not optimal.

According to Russell & Norvig (2009), the time complexity of DFS is $O(b^m)$, where where m is the maximum depth of any node and b is the branching factor. The space complexity of DFS is $O(bm)$ (Russell & Norvig 2009).

2.2.2 Applications of Depth First Search

Some applications of DFS are listed below (GeeksforGeeks 2017).

1. **Topology sorting:** scheduling jobs from the given dependencies among jobs (instruction scheduling, cell evaluation in spreadsheets, etc).
2. **Path finding**

3. **Detecting cycle in a graph**

2.3 Best First Search

In Best First Search (BestFS), a node is selected for expansion based on an evaluation function, $f(n)$ (Russell & Norvig 2009). The evaluation function is considered as a cost estimate, which means that the node with the lowest score is explored first (Russell & Norvig 2009). In this case, evaluation function just a heuristic function $f(n) = h(n)$, where $h(n)$ is an estimated cost of the path from the state at node n to a goal state (Russell & Norvig 2009). An example of node expansion of BestFS is shown in Figure 4.

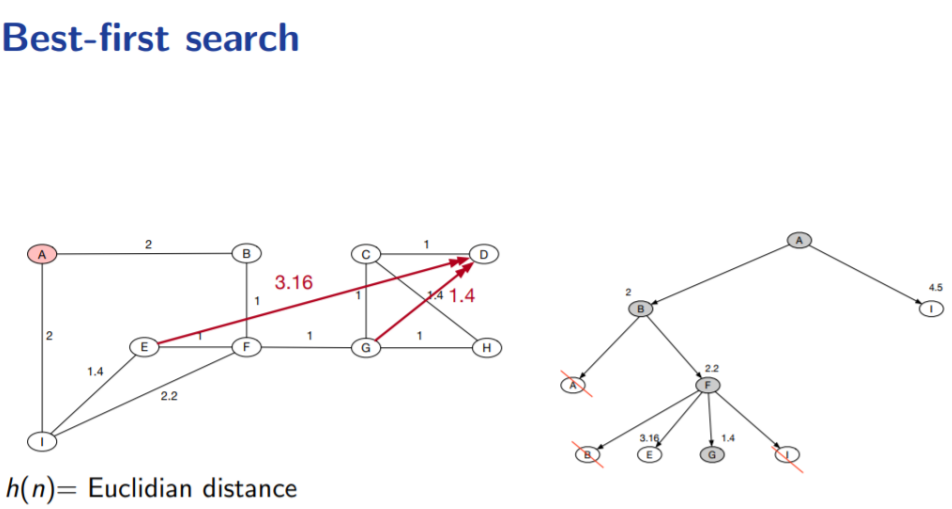

Figure 4: Node expansion in Best First Search. (Toniolo 2017*b*).

2.3.1 Analysis

According to Russell & Norvig (2009), BestFS is complete in finite spaces, but not in infinite ones. The time and space complexity is $O(b^m)$, where m is the maximum depth of the search space (Russell & Norvig 2009). Russell & Norvig (2009) also states that the algorithm is not optimal since it only tries to get as close to the goal as it can (greedy).

2.3.2 Applications of Best First Search

Some applications of BestFS are listed below (Mussmann & See n.d.).

1. **Shortest path problems:** find the shortest path from initial position to target position.
2. **Robotics:** domestic robots, search and rescue robots, commercial robots, etc.
3. **Route planning**

2.4 A* Search

A* search evaluates nodes by combining $g(n)$, the cost to reach node n, and $h(n)$, an estimated cost of the path from the state at node n to a goal state (Russell & Norvig 2009).

$$f(n) = g(n) + h(n) \tag{1}$$

In this case, the node with the lowest $f(n)$ will be expanded first. An example of node expansions using A* can be found in Figure 5.

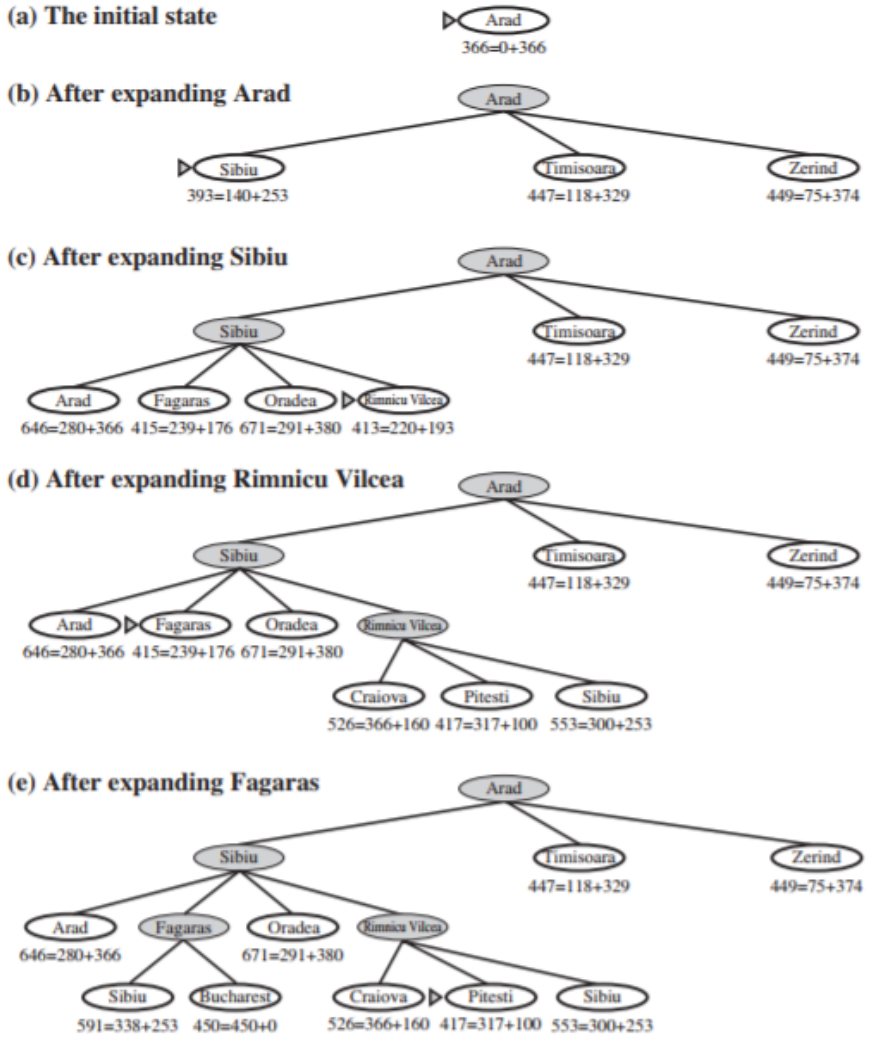

Figure 5: Node expansion in A* Search. (Russell & Norvig 2009).

2.4.1 Analysis

According to (Russell & Norvig 2009), A* search is both complete and optimal. A* is optimal if $h(n)$ is an admissible heuristic, which means that it never overestimates the cost to reach the goal (Russell & Norvig 2009). A* search's time and space complexity is $O(b^d)$, where b is branching factor, and d is the depth of the solution (Russell & Norvig 2009).

2.4.2 Applications of A* Search

Some applications of A* search are listed below.

1. **Finding the best path in a directed acyclic graph with associated edge scores (Kelly & Labute 1996)**
2. **Route planning**

3 Part 1

3.1 Design

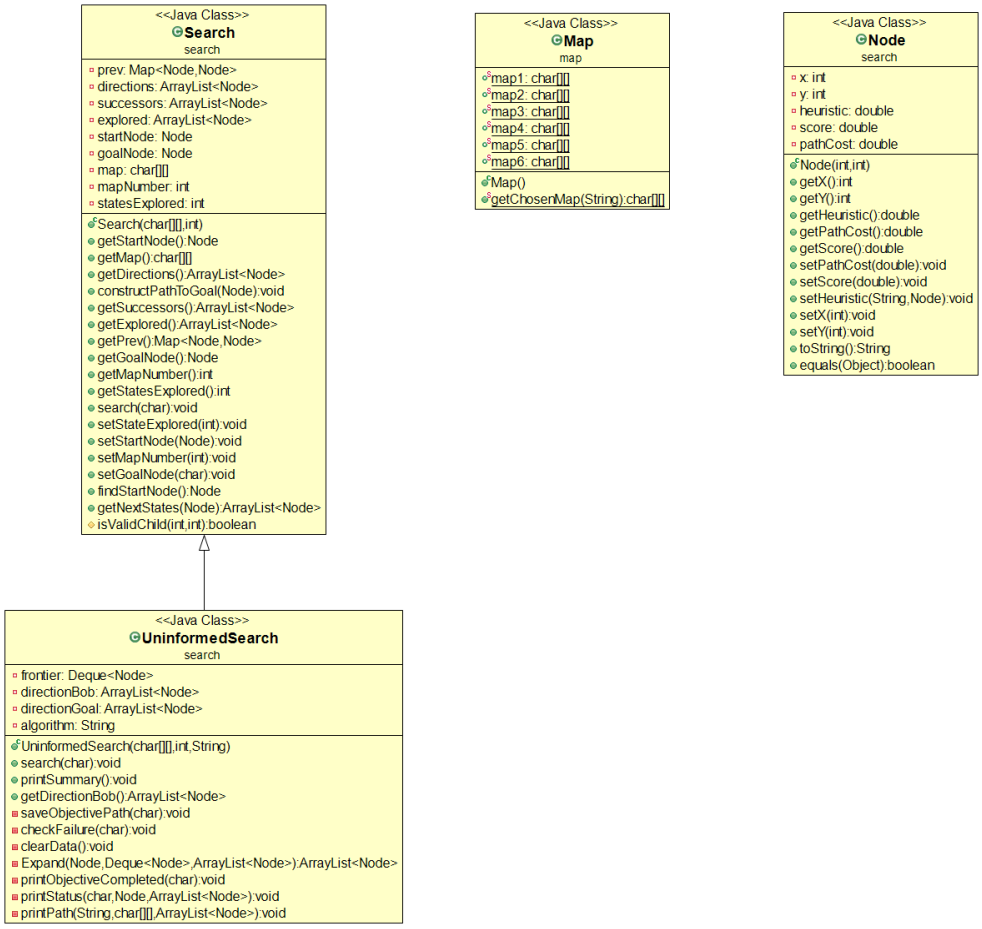

Figure 6: Class diagram for part 1.

3.1.1 Problem Formulation

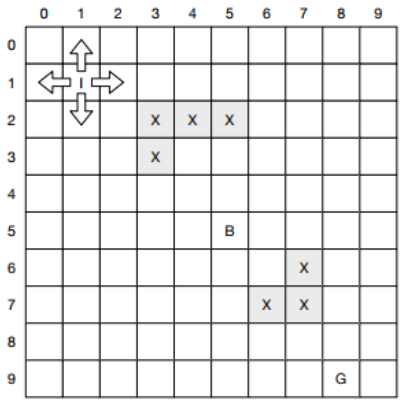

Figure 7: Example map (Toniolo 2017*a*).

State space: The state space is the positions in the map (see Figure 7). In the implementation, each position is represented by a Node object as shown in Figure 6.

Initial State: Robot is at position marked as I as shown in Figure 7. This information is stored in the ***startNode*** attribute of ***Search*** class (see Figure 6).

Goal: Find the path from I to B, then from B to G avoiding obstacles. The goal is stored in ***goalNode*** attribute of ***Search*** class as illustrated in Figure 6.At the start, the goal is the position marked as B. Once the algorithm found a path to B, the goal changes to position marked as G.

Successor functions: Generates nodes representing the cells adjacent to the robot's position. The successor function is implemented as ***getNextStates()*** method in ***Search*** class (see Figure 6). The nodes representing potential next moves are added to an ***ArrayList*** called ***nextStates***.

```
protected boolean isValidChild(int x, int y) {
    // validate if the move is legal or not (out of bound, blocked, etc.)
    return !(x < 0 || x >= map.length || y < 0 || y >= map[0].length) &&
        (map[x][y] != 'X');
  }
```

Listing 1: The code fragment used to validate next states.

Actions: The robot can move one cell at a time to the adjacent cells in the north, south, east, or west directions. Some of the positions are blocked by obstacles marked by X, which means that the robot cannot move to that position (see Figure 7). The method used to validate the actions is ***isValidChild()*** method in ***Search*** class as shown in Figure 6.

This method is used to validate the nodes that represent the cells adjacent to the current position on the map. The validation process includes checking if node's position is still in the map or not (index out of bound check), and identifying if the node is blocked or not (see Listing 1).

Path cost: Since the maps are in the form of 10x10 evenly laid out grid as in Figure 7, the path cost to move from one cell to another is 1. The value of path cost is stored in ***pathCost*** attribute of ***Node*** class (see Figure 6).

3.1.2 Search Algorithms Implementation

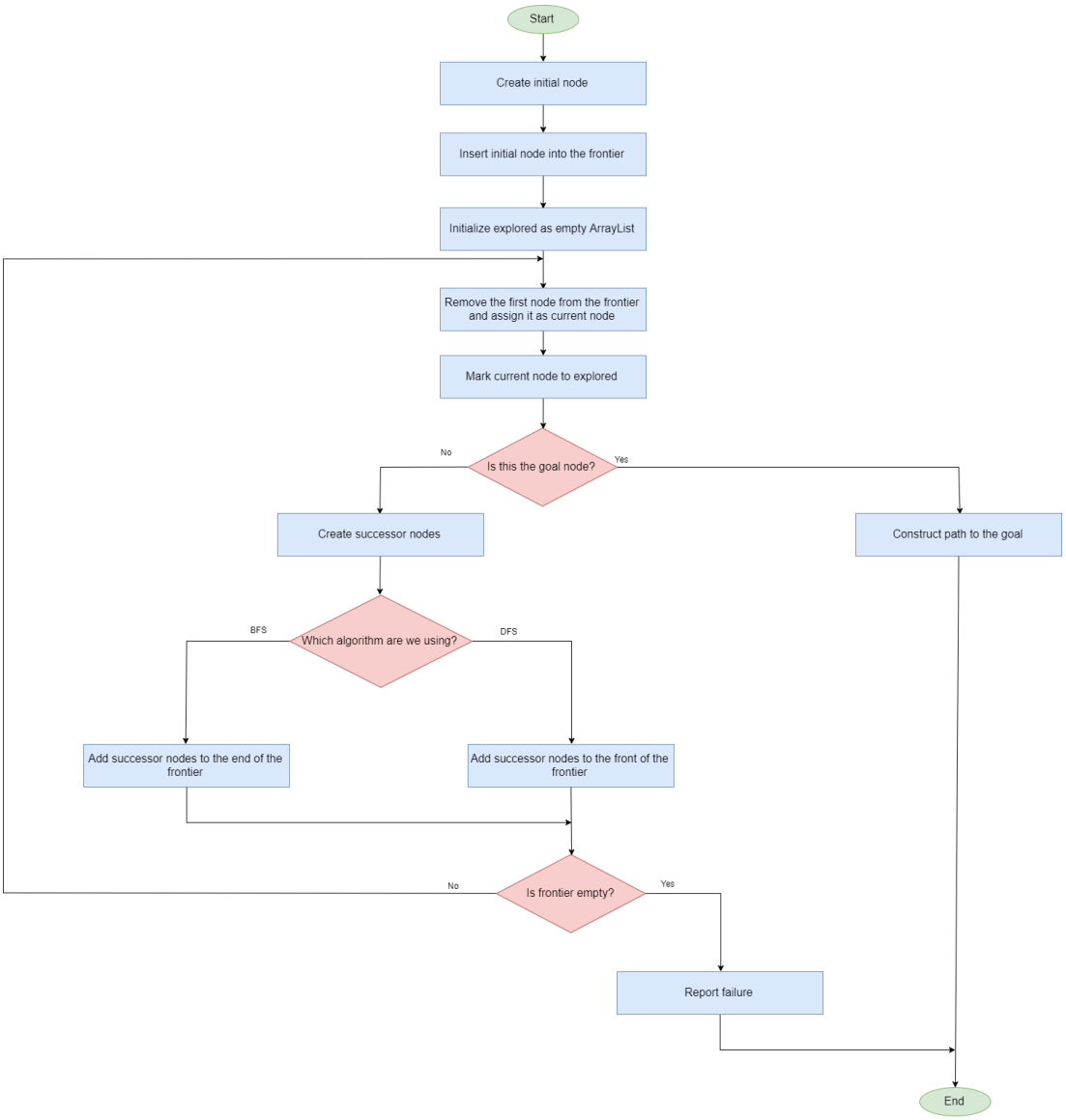

Figure 8: Flowchart showing the execution of the program for part 1.

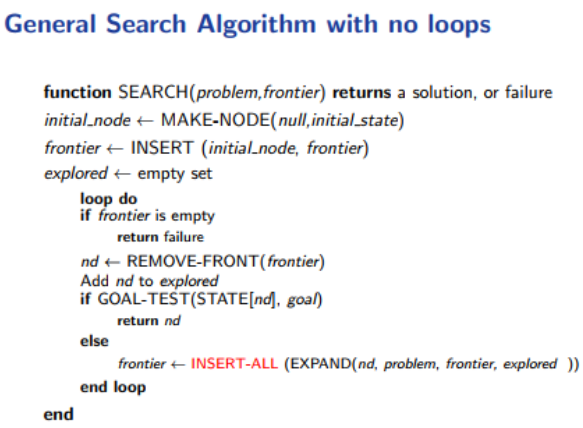

Figure 9: General Search algorithm (Toniolo 2017*b*).

```
for(Node node : successors) {
  prev.put(node, currentNode);
  if (algorithm.equals("BFS")) {
  frontier.addLast(node);
  } else {
  frontier.addFirst(node);
  }
}
```

Listing 2: The code fragment used to add successor nodes to the frontier based on chosen algorithm.

The implementation of search algorithms is based on the pseudo-code given in the lecture slides by Toniolo (2017*b*) (see Figure 9). The method that is used to perform search operation is ***search()*** method in ***UninformedSearch*** class (see Figure 6). The functionalities of Breadth First Search and Depth First Search are almost identical to each other. However, the main difference is that BFS inserts successor nodes at the end of the frontier, whereas DFS inserts successor nodes at the front of the frontier. This behavior is illustrated in the code fragment in Listing 2. Both algorithms use Deque data structure to store the frontier since it can be used as LIFO (DFS) or FIFO (BFS) queue. The flowchart illustrating the execution cycle of the program for part 1 can be found in Figure 8.

```java
public ArrayList<Node> getNextStates(Node node) {
    int x = node.getX();
    int y = node.getY();
    ArrayList<Node> nextStates = new ArrayList<Node>();
    // get the potential next moves (North, South, East, West)
    // North
        if(isValidChild(x - 1, y)) {
          nextStates.add(new Node(x - 1, y));
        }

        // West
        if(isValidChild(x, y - 1)) {
          nextStates.add(new Node(x, y - 1));
        }

        // East
        if(isValidChild(x, y + 1)) {
          nextStates.add(new Node(x, y + 1));
        }

        // South
    if(isValidChild(x + 1, y)) {
      nextStates.add(new Node(x + 1, y));
      }

    return nextStates;
  }
```

Listing 3: The code fragment used to add new states.

For this part, the new states are added in the following order: South, East, West, and North. This design decision was chosen because both BFS and DFS are considered to be blind search, which means that they do not use heuristic to choose the next state to explore. The decision to add the states in fixed order is to keep the search result consistent during the testing (always get the same result when executing the search). The code used to add new states is shown in Listing 3.

3.1.3 Path Construction

```java
public void constructPathToGoal(Node currentNode) {
    // construct path to goal by backtracking from goal to initial position
    for(Node node = currentNode; node != null; node = prev.get(node)) {
          directions.add(node);
      }
    Collections.reverse(directions);
}
```

Listing 4: The code fragment used to construct the path from initial position to goal position.

As stated in the requirements, each algorithm can construct the path it has found from the starting point to Bob, and to the goal. This was done by storing the mapping of each

node to its parent node in the ***prev*** attribute of ***Search*** class. Once the algorithm found the goal, it uses the code in Listing 4 to construct the path from initial position to goal position by using the information stored in ***prev*** attribute to backtrack from the goal node to initial node, and reverse the path in order to get the path from initial position to goal position.

3.2 Examples and Testing

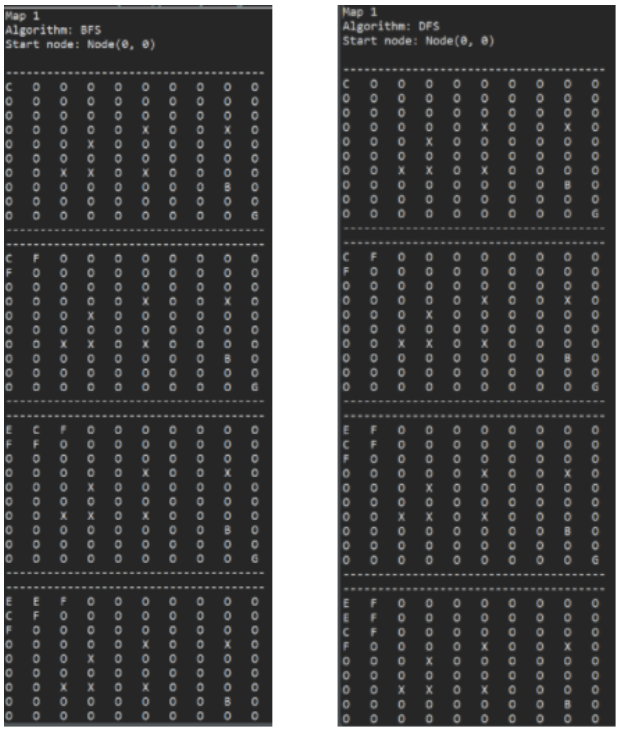

Figure 10: Printout showing current node (C), states expanded (E), and the frontier (F) when using BFS and DFS in map 1.

To demonstrate that the implementation works correctly, the algorithms print out current node (C), states expanded (E), and the frontier (F) at every step. An example of printout from BFS and DFS can be found in Figure 10.

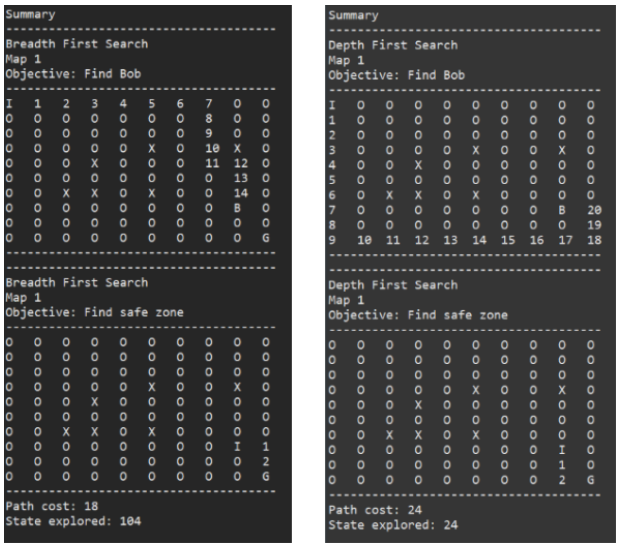

Figure 11: Printout showing paths constructed by BFS and DFS in map 1.

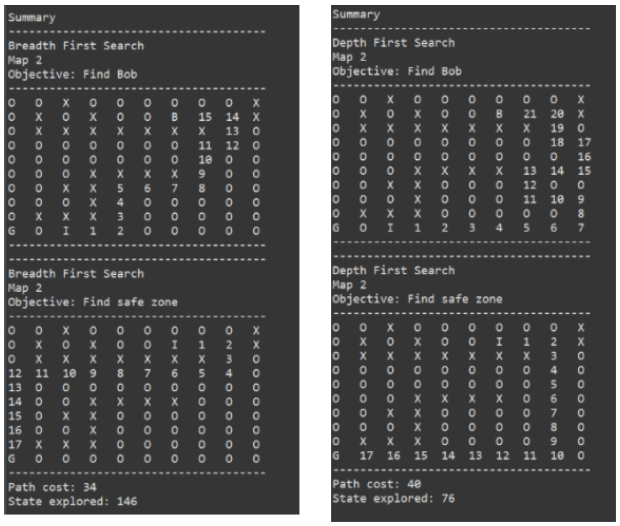

Figure 12: Printout showing paths constructed by BFS and DFS in map 2.

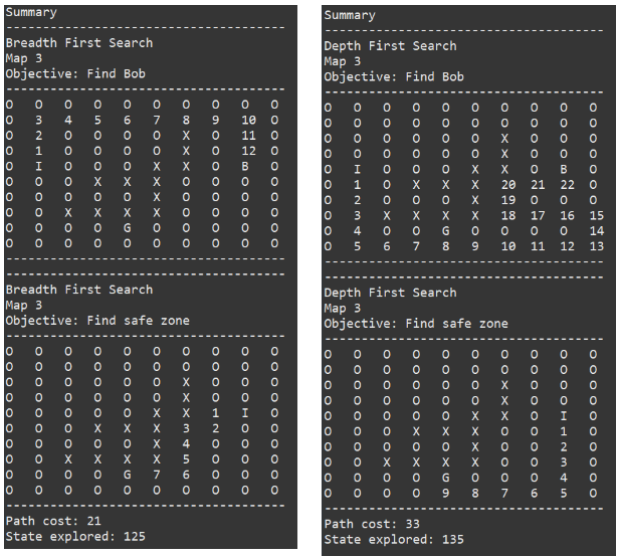

Figure 13: Printout showing paths constructed by BFS and DFS in map 3.

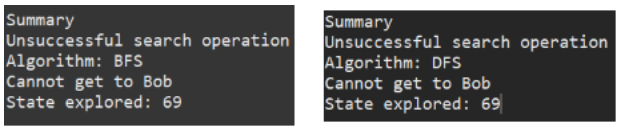

Figure 14: Printout showing failure of search operation in map 4.

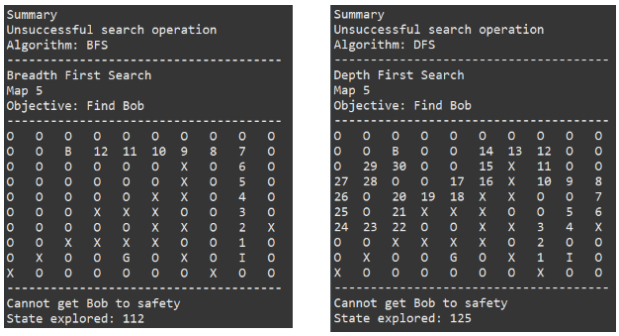

Figure 15: Printout showing failure of search operation in map 5.

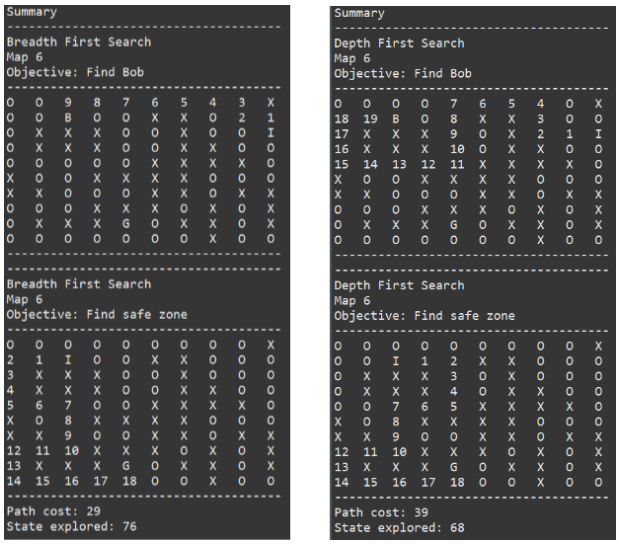

Figure 16: Printout showing paths constructed by BFS and DFS in map 6.

The paths constructed by BFS and DFS in all maps provided can be found in Figure 11, Figure 12, Figure 13, Figure 14, Figure 15, and Figure 16. (the numbers represent the order in which the robot navigates to a particular position on the maps).

Map 1	Search states visited	Path cost	Remarks
Breadth First Search	104	18	
Depth First Search	24	24	

Table 1: BFS and DFS performance in map 1

Map 2	Search states visited	Path cost	Remarks
Breadth First Search	146	34	
Depth First Search	76	40	

Table 2: BFS and DFS performance in map 2

Map 3	Search states visited	Path cost	Remarks
Breadth First Search	125	21	
Depth First Search	135	33	

Table 3: BFS and DFS performance in map 3

Map 4	Search states visited	Path cost	Remarks
Breadth First Search	69	-	Cannot get to Bob
Depth First Search	69	-	Cannot get to Bob

Table 4: BFS and DFS performance in map 4

Map 5	Search states visited	Path cost	Remarks
Breadth First Search	112	-	Manged to find Bob, but cannot find a path to goal position
Depth First Search	125	-	Manged to find Bob, but cannot find a path to goal position

Table 5: BFS and DFS performance in map 5

Map 6	Search states visited	Path cost	Remarks
Breadth First Search	76	29	
Depth First Search	68	39	

Table 6: BFS and DFS performance in map 6

The summary of BFS and DFS performance in all maps can be found in Table 1, Table 2, Table 3, Table 4, Table 5, and Table 6.. The evaluation of both algorithm can be found in Section 3.3.

3.3 Evaluation

Overall, the implementation of BFS and DFS managed to meet all of the requirements. DFS has lower number of search states visited most cases, while BFS has lower path cost in all maps. These results corresponds with the theoretical explanation of both algorithms. Since DFS starts at root node and explores as far as possible along each branch of graph (LIFO behavior with the queue), it is possible that there will be numerous unvisited nodes when DFS found the goal position. As a result, the path that DFS found may not be optimal. On the other hand, BFS starts at root node and explores the neighbor nodes first, before moving to the next level of the graph (FIFO behavior with the queue). This means that when BFS found the goal, it would have explored all the nodes at higher depth. Therefore, it can always construct an optimal path to the goal position. It is worth mentioning that both algorithms cannot complete the search operation in map 4 and map 5. This is expected because all the paths to Bob are blocked in map 4, while all the paths to goal position are blocked in map 5. Due to time constraints, it is not possible to test all combinations of order that new states are added. There may be a combination that can lead to better performance for both algorithms.

4 Part 2

4.1 Design

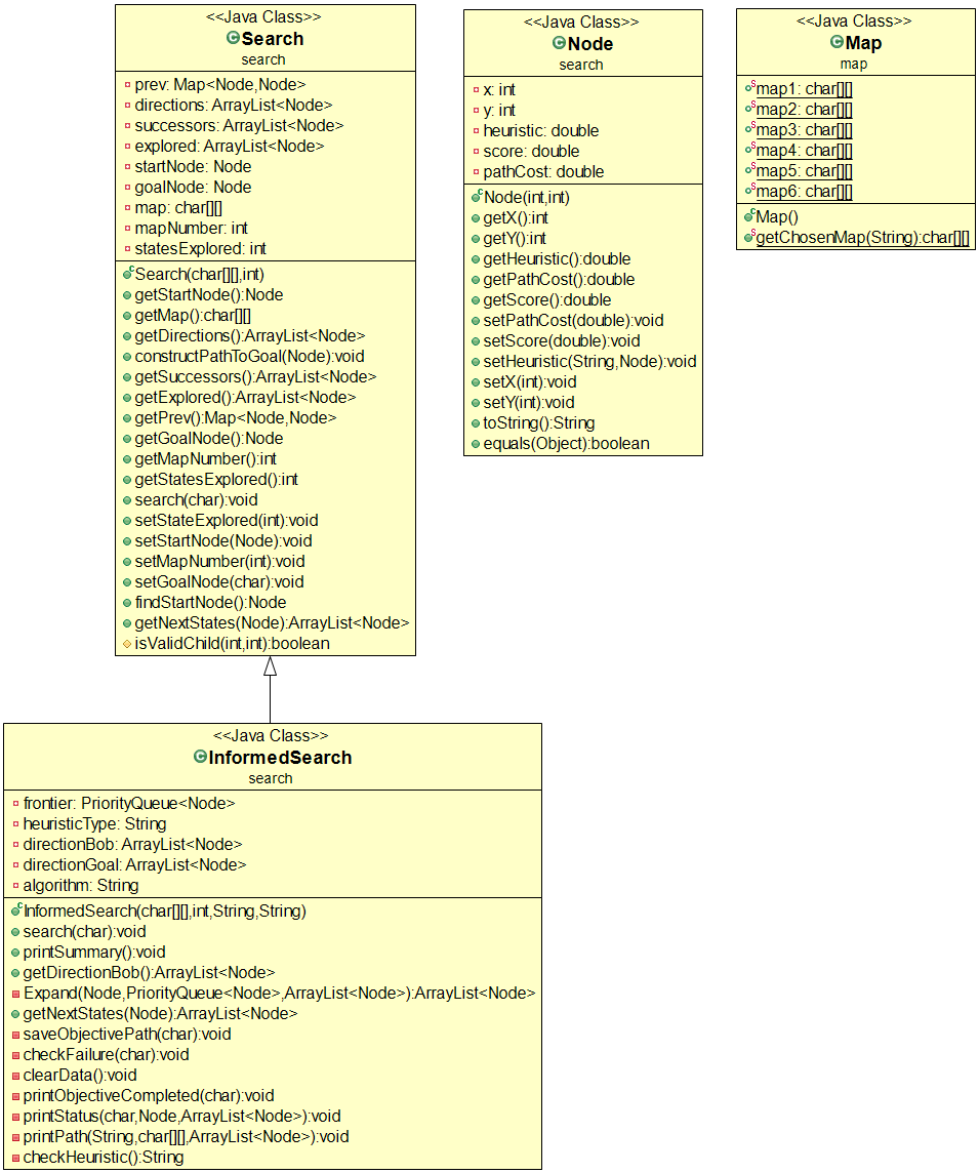

Figure 17: Class diagram for part 2.

4.1.1 Manhattan Distance

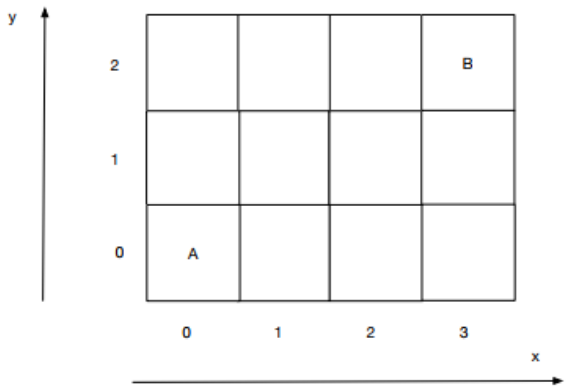

Figure 18: Grid-based distance between two positions.

Manhattan distance is the distance between two positions in a grid (see Figure 18). It can be calculated with the equation:

$$distance = |X_A - X_B| + |Y_A - Y_B| \tag{2}$$

The value of Manhattan distance is stored in ***heuristic*** attribute of ***Node*** class (see Figure 17). The calculation of Manhattan distance is shown in Listing 5.

```
this.heuristic = Math.abs(goalNode.getX() - this.getX()) +
        Math.abs(goalNode.getY() - this.getY());
```

Listing 5: The code fragment used to calculate Manhattan distance.

4.1.2 Euclidean Distance

Euclidean distance is the straight line distance between two positions. I can be calculated with the equation:

$$distance = \sqrt{(X_A - X_B)^2 + (Y_A - Y_B)^2} \tag{3}$$

The value of Euclidean distance is stored in ***heuristic*** attribute of ***Node*** class. The calculation of Euclidean distance is shown in Listing 6.

```
this.heuristic = Math.sqrt(Math.pow(goalNode.getX() - this.getX(), 2) +
        Math.pow(goalNode.getY() - this.getY(), 2));
```

Listing 6: The code fragment used to calculate Euclidean distance.

4.1.3 Combining the Heuristics

In this case, the heuristic is calculated by taking the maximum value between Manhattan distance and Euclidean distance.

$$h(h) = max\{h_1(n), h_2(n)\} \tag{4}$$

The value of heuristic is stored in ***heuristic*** attribute of ***Node*** class. The calculation of heuristic is shown in Listing 7.

```
// choose the maximum between Manhattan distance and Euclidean distance
    double manDist = Math.abs(goalNode.getX() - this.getX()) +
        Math.abs(goalNode.getY() - this.getY());
    double euclidDist = Math.sqrt(Math.pow(goalNode.getX() - this.getX(),
2) +
        Math.pow(goalNode.getY() - this.getY(), 2));
    this.heuristic = Math.max(manDist, euclidDist);
```

Listing 7: The code fragment used to calculate maximum of Manhattan distance and Eucildean distance.

4.1.4 Search Algorithms Implementation

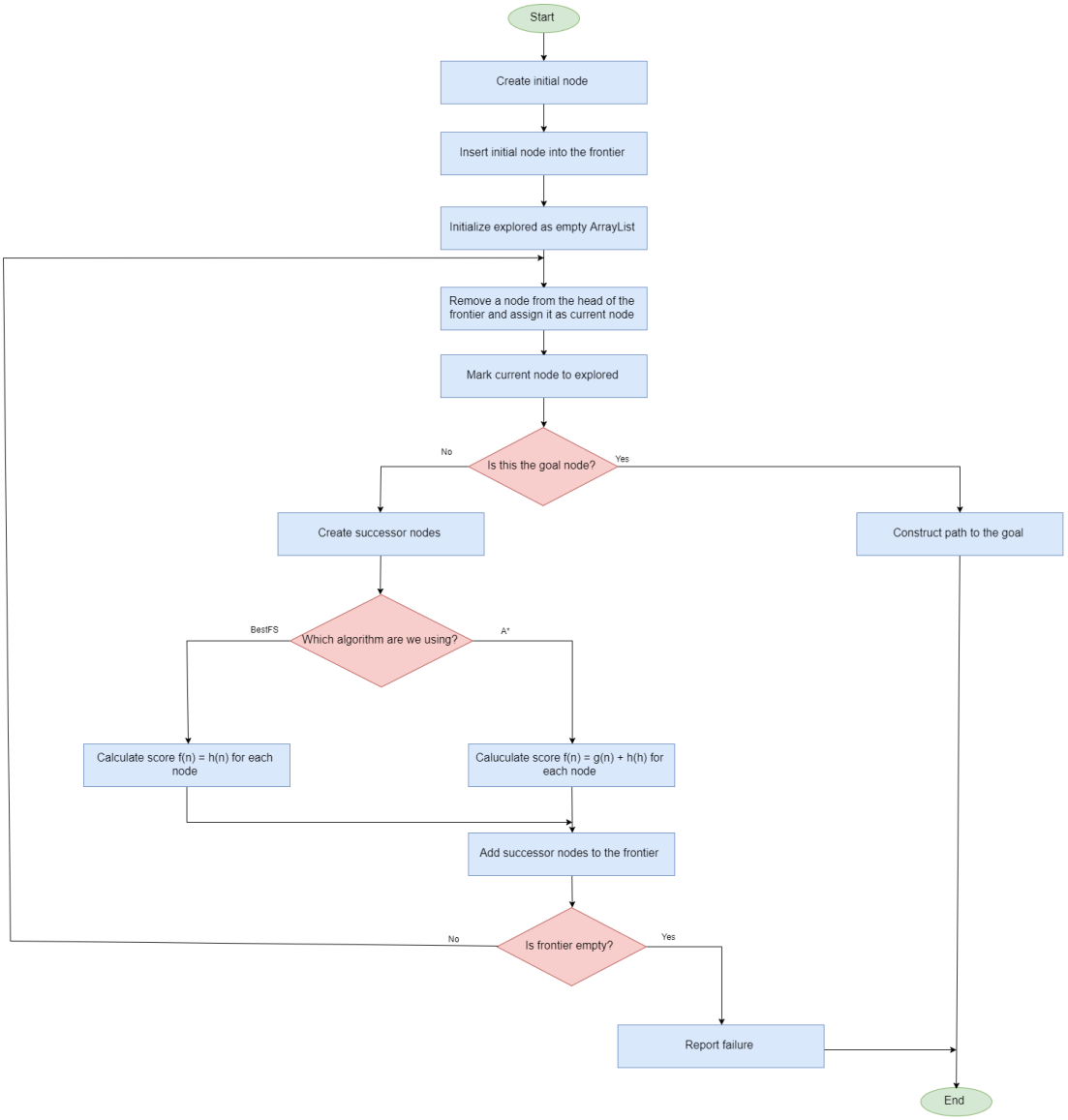

Figure 19: Flowchart showing the execution of the program for part 2.

```
if(isValidChild(x - 1, y)) {
  Node up = new Node(x - 1, y);
  up.setPathCost(node.getPathCost() + 1);
  up.setHeuristic(heuristicType, goalNode);
  if (algorithm.equals("BestFS")) {
  up.setScore(up.getHeuristic());
  } else {
  up.setScore(up.getHeuristic() + up.getPathCost());
  }
  nextStates.add(up);
}
```

Listing 8: The code fragment used to assign score to a successor node based on chosen algorithm.

The implementation of search algorithms is based on the pseudo-code of general search given in the lecture slides by Toniolo (2017*b*). The method that is used to perform search operation is ***search()*** method in ***InformedSearch*** class (see Figure 17). The functionalities of Best First Search (BestFS) and A* are almost the same as each other. However, the main difference is that BestFS assigns score $f(n) = h(n)$ to the successor nodes, whereas A* search assigns score $f(n) = g(n) + h(n)$ to the successor nodes. g(n) is the cost of the path from the start to the node n, and h(n) is the estimated cost of the path from the state at node n to the goal. This behavior is implemented in ***getNextStates*** method of ***InformedSearch*** class. An example code that was used to assign a score to a successor node located north of current position is shown in Listing 8. Both algorithms use priority queue data structure to store the frontier since it can be used to give priority based on the score of each node. Consequently, both algorithms can get the node with the lowest score with ***frontier.poll()*** method. The implementation of frontier PriorityQueue is shown in Listing 9. The flowchart illustrating the execution cycle of the program for part 2 can be found in Figure 19. Like part 1, Best First Search and A* search can construct the path they have found from the starting point to Bob, and to the goal. The method used to construct the path is the same method that was used in part 1 (see Section 3.1.3).

```
  frontier = new PriorityQueue<Node>(new Comparator<Node>() {
      public int compare(Node n1, Node n2) {
        if (n1.getScore() < n2.getScore()) {
          return -1;
        }
        if (n1.getScore() > n2.getScore()) {
          return 1;
        }
        return 0;
      }
    });
```

Listing 9: The code fragment used to initialize frontier PriorityQueue.

4.2 Examples and Testing

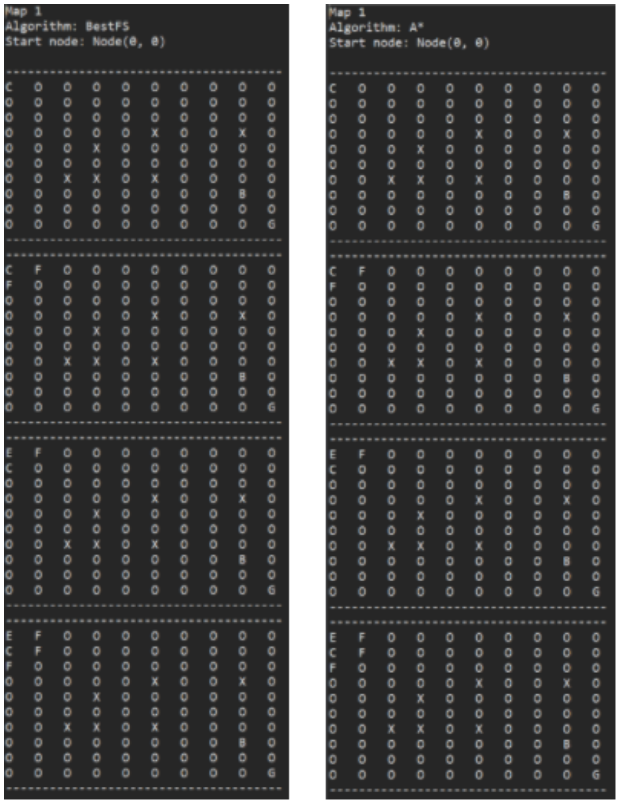

Figure 20: Printout showing current node (C), states expanded(E), and the frontier (F).

To demonstrate that the implementation works correctly, the algorithms print out current node (C), states expanded (E), and the frontier (F) at every step. An example of printout from BestFS and A* can be found in Figure 20.

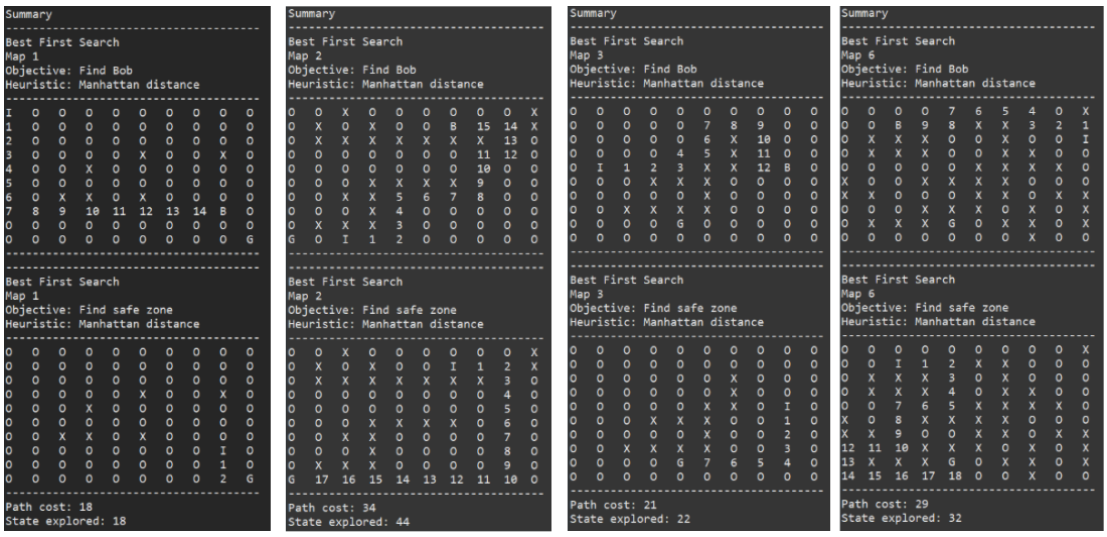

Figure 21: Printout showing paths constructed by Best First Search using Manhattan distance as heuristic.

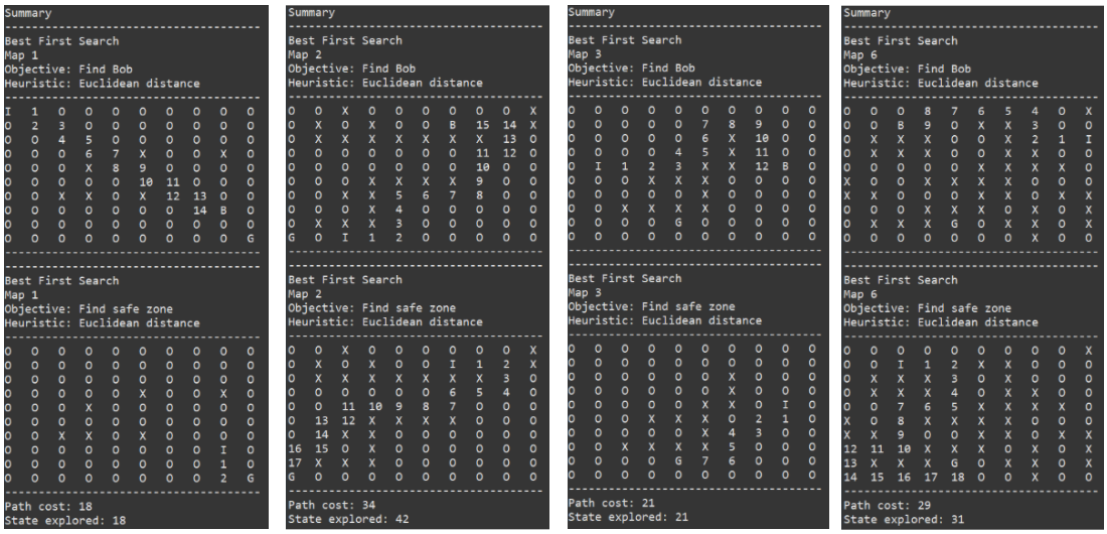

Figure 22: Printout showing paths constructed by Best First Search using Euclidean distance as heuristic.

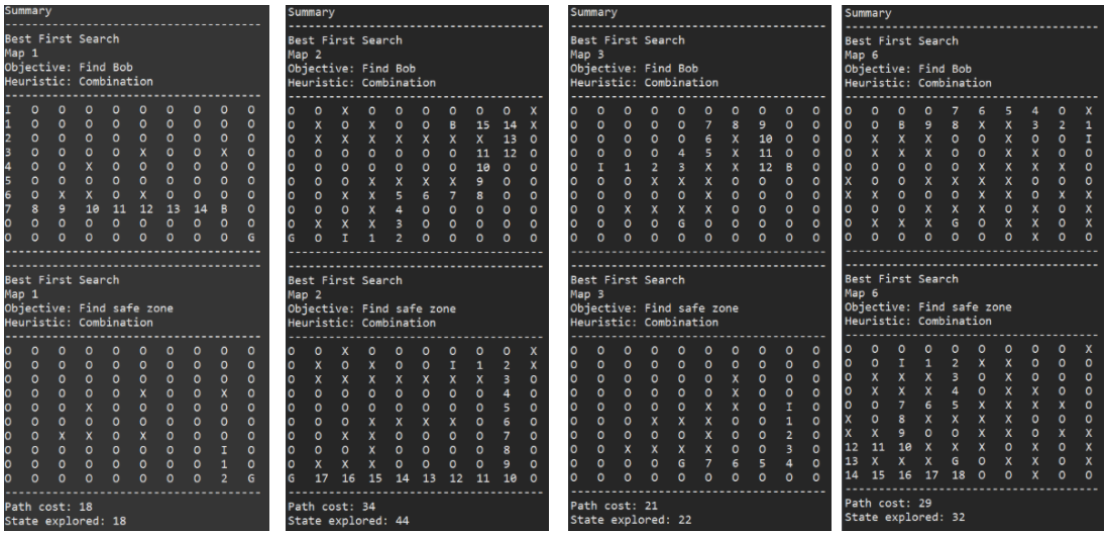

Figure 23: Printout showing paths constructed by Best First Search using combination of Manhattan distance and Euclidean distance as heuristic.

The paths constructed by Best First Search algorithm in map 1, 2, 3, and 6 are shown in Figure 21, Figure 22, and Figure 23. Map 4 and map 5 were not used because it is impossible to successfully complete the search operation (see Section 3.3).

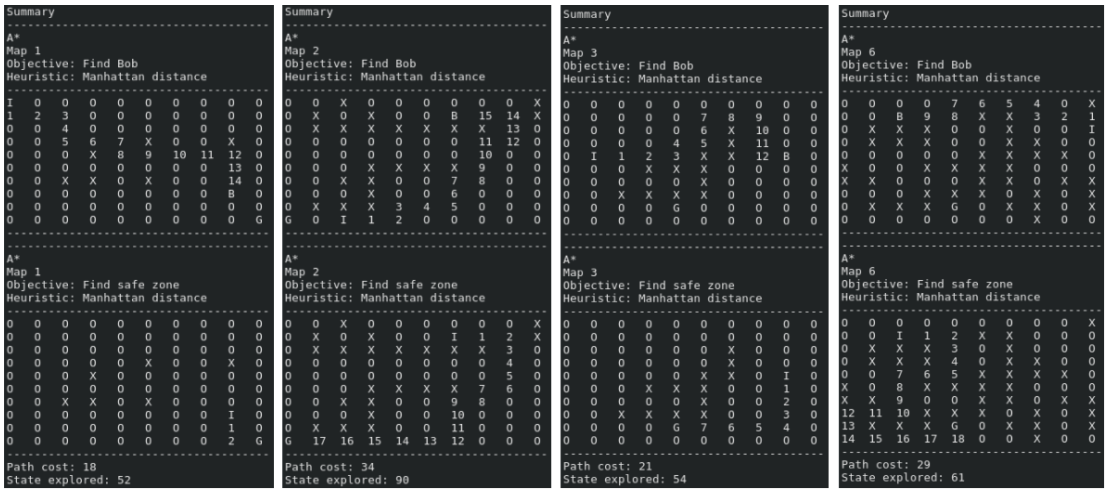

Figure 24: Printout showing paths constructed by A* search using Manhattan distance as heuristic.

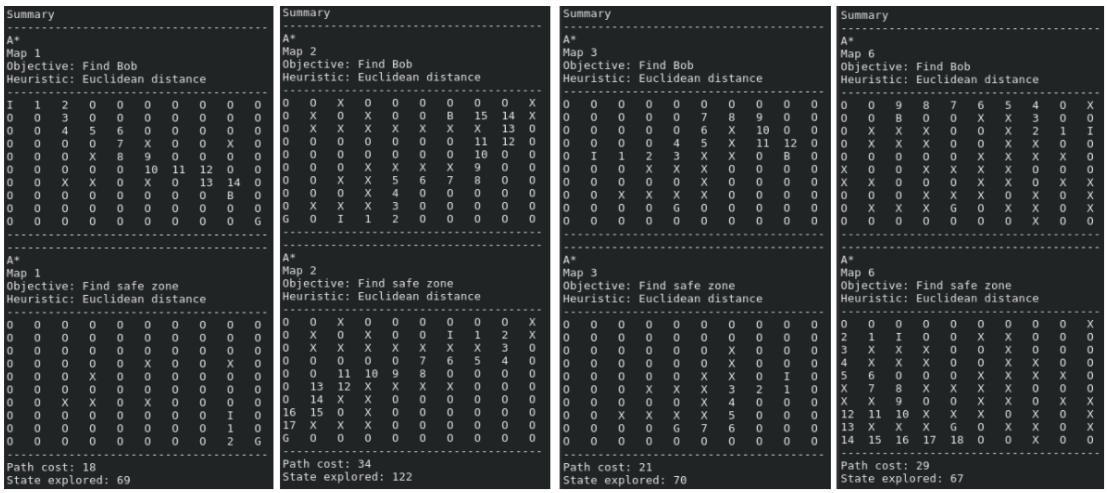

Figure 25: Printout showing paths constructed by A* search using Euclidean distance as heuristic.

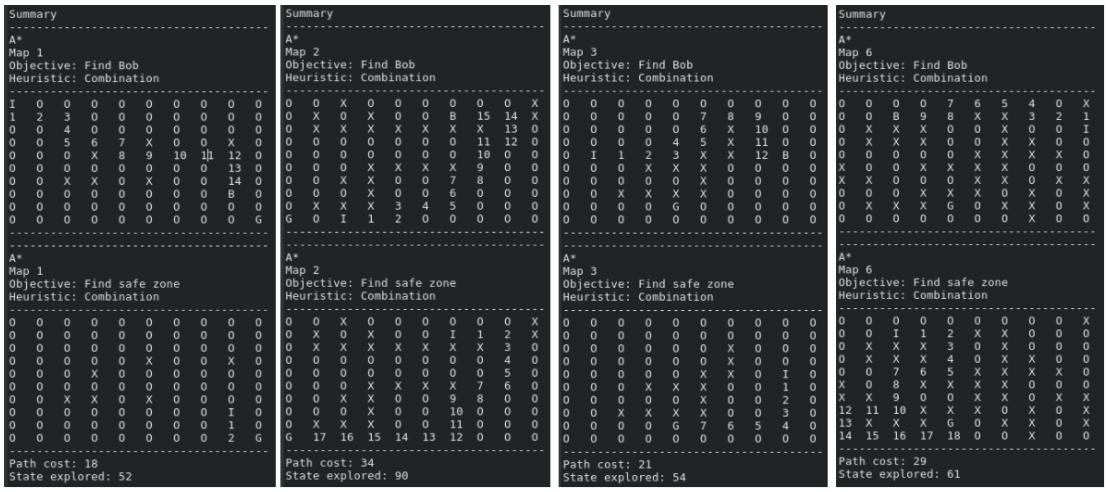

Figure 26: Printout showing paths constructed by A* search using combination of Manhattan distance and Euclidean distance as heuristic.

The paths constructed by A* algorithm in map 1, 2, 3, and 6 are shown in Figure 24, Figure 25, and Figure 26.

Map 1	Search states visited	Path cost	Remarks
Best First Search (M)	18	18	
A* Search (M)	52	18	
Best First Search (E)	18	18	
A* Search (E)	69	18	
Best First Search (all)	18	18	
A* Search (all)	52	18	

M - Manhattan distance
E - Euclidean distance
all - Combines Manhattan distance and Euclidean distance

Table 7: Best First Search and A* search performance in map 1

Map 2	Search states visited	Path cost	Remarks
Best First Search (M)	44	34	
A* Search (M)	90	34	
Best First Search (E)	42	34	
A* Search (E)	122	34	
Best First Search (all)	44	34	
A* Search (all)	90	34	

M - Manhattan distance
E - Euclidean distance
all - Combines Manhattan distance and Euclidean distance

Table 8: Best First Search and A* search performance in map 2

Map 3	Search states visited	Path cost	Remarks
Best First Search (M)	22	21	
A* Search (M)	54	21	
Best First Search (E)	21	21	
A* Search (E)	70	21	
Best First Search (all)	22	21	
A* Search (all)	54	21	

M - Manhattan distance
E - Euclidean distance
all - Combines Manhattan distance and Euclidean distance

Table 9: Best First Search and A* search performance in map 3

Map 6	Search states visited	Path cost	Remarks
Best First Search (M)	32	29	
A* Search (M)	61	29	
Best First Search (E)	31	29	
A* Search (E)	67	29	
Best First Search (all)	32	29	
A* Search (all)	61	29	

M - Manhattan distance
E - Euclidean distance
all - Combines Manhattan distance and Euclidean distance

Table 10: Best First Search and A* search performance in map 6

The summary of BFS and DFS performance in all maps can be found in Table 7, Table 8, Table 9, and Table 10. Map 4 and map 5 were left out because the tests in part 1 suggest that it is impossible to find a solution for those maps.

4.3 Evaluation

For this part, Best First Search (BestFS) and A* search are implemented according to the requirements. BestFS seems to have lower number of states explored than A*. This is expected since BestFS only tries to get as close to the goal as it can without considering path cost like A*. The two algorithms also have the same path cost in every map, even though BestFS is not optimal. This is quite surprising since there is no implementation of tie-breaking strategy in case the successor nodes have the same score. The fact that BestFS managed to choose an optimal path is due to pure chance in this case. It is worth mentioning that there are no different in path cost when using Manhattan distance and Euclidean distance as heuristic. However, the performance summary suggests that A* has lower number of search states visited when using Manhattan distance as heuristic. When combination of two heuristics is used, it seems that Manhattan distance dominates Euclidean distance (the number of search states visited are the same as when Manhattan distance was used). In this case, A* is a better option in search and rescue operation because it is both complete and optimal, whereas BestFS is complete (in finite grid space), but not optimal. Manhattan distance is suitable for A* in this search and rescue situation because it minimizes the number of search states visit, which means that the agent can finish the search procedure faster. There might be other heuristics that can lower the number of states visited even further. In the future, it may be worth exploring to see which heuristic can be used to improve the performance of this system.

5 Part 3

5.1 Design

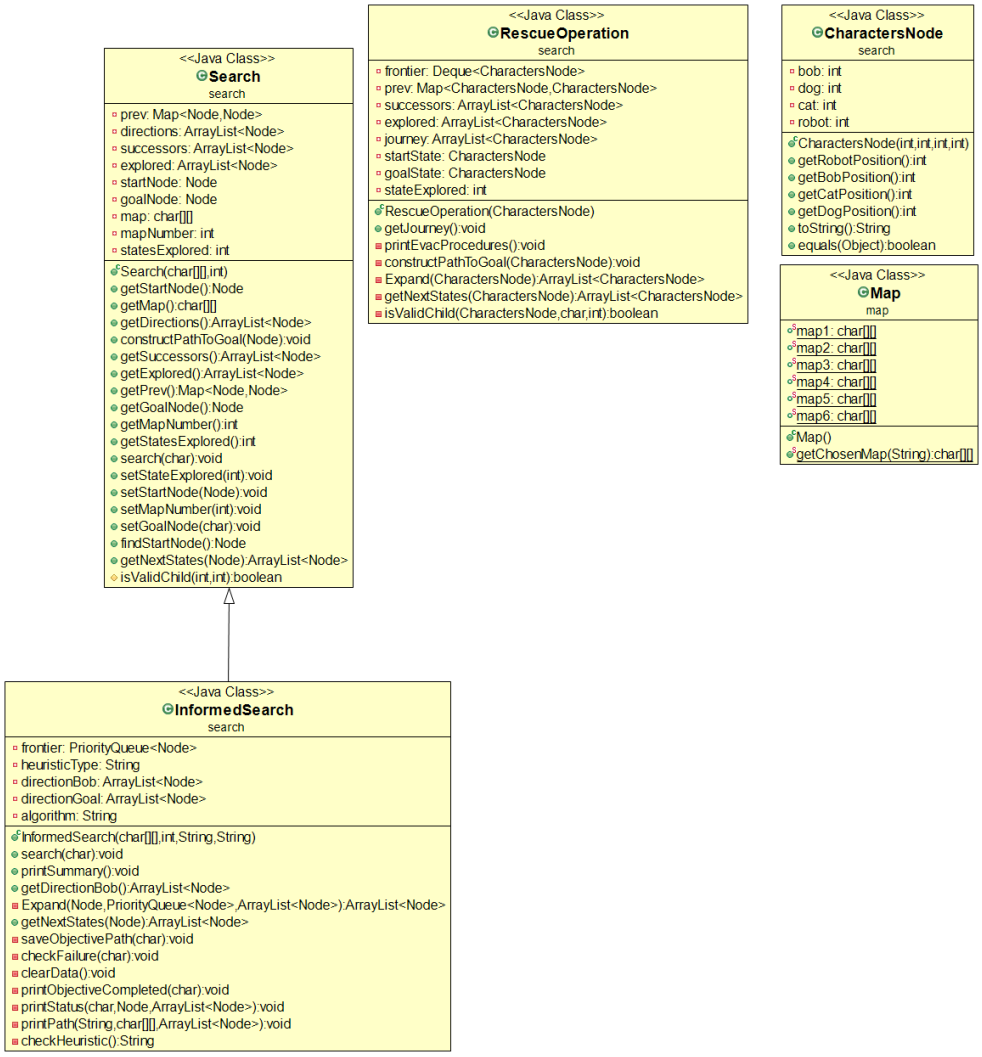

Figure 27: Class diagram for part 3.

5.1.1 Problem Formulation

State space: The state space is the positions of the robot and the victims (position B or position G). In the implementation, the position of robot and victims are stored in ***CharactersNode*** class (see Figure 27). For simplicity robot and the survivors will be referred to as characters. In this case, a character attribute in the class will be 0 if a particular character is at position B and the value will be 1 if a particular character is at position G. For example, the method ***getBobPosition()*** of ***CharactersNode*** class returns 0 if Bob is at position B.

Initial State: Robot, Bob, Dog, and Cat are at position marked as B. This is represented by ***CharacterNode*** object with all characters at position 0.

Goal: Find the sequence of actions that allows the robot to bring the survivors to safety. This is represented by ***CharacterNode*** object with all characters at position 1.

Successor functions: Generates nodes representing state of each character. The successor function is implemented as ***getNextStates()*** method in ***RescueOperation*** class (see Figure 27). The nodes representing potential next actions are added to an ***ArrayList*** called ***nextStates***.

Actions: The robot can only transport one of the victims at a time to the safe goal cell G. The problem is that Bob cannot stay alone with the dog, and the dog cannot stay alone with the cat. This implies that the following states are illegal:

- only Bob and Dog in Cell B
- only Dog and Cat in Cell B
- only Bob and Dog in Cell G
- only Dog and Cat in Cell G

The method used to validate the actions is ***isValidChild()*** method in ***RescueOperation*** class as shown in Figure 27. The method checks if the robot is in the same position as the character that is going to be moved (survivors cannot move by themselves). It also checks if the move leads to an illegal state or not. The code can be found in Listing 10.

Path cost: The path cost is the number of times that the robot move the victims between position B and position G.

```
private boolean isValidChild (CharactersNode characters, char character, int
    posChange) {
    int robotPos = characters.getRobotPosition();
    int bobPos = characters.getBobPosition();
    int catPos = characters.getCatPosition();
    int dogPos = characters.getDogPosition();
    if (character == 'r') {
      // check if robot can be moved
      int newPos = robotPos + posChange;
      boolean invalidPos = (bobPos == dogPos && newPos != bobPos) ||
          (catPos == dogPos && newPos != catPos);
      if (!invalidPos) {
        return true;
      } else {
        return false;
      }
    } else if(character == 'b' && robotPos == bobPos) {
      // check if Bob can be moved
      int newPos = robotPos + posChange;
      boolean invalidPos = (catPos == dogPos && newPos != catPos);
      if (!invalidPos) {
        return true;
      } else {
        return false;
      }
    } else if (character == 'c' && robotPos == catPos) {
      // check if the cat can be moved
      int newPos = robotPos + posChange;
      boolean invalidPos = (bobPos == dogPos && newPos != bobPos);
      if (!invalidPos) {
        return true;
      } else {
        return false;
      }
    } else if (character == 'd' && robotPos == dogPos) {
      // check if the dog can be moved
      return true;
    } else {
      return false;
    }
  }
```

Listing 10: The code fragment used to validate next states of the chracters.

5.1.2 Implementation

In this part, the algorithm that is used to find the path to Bob and to goal position is A*. As discussed in Section 4.3, A* is the best algorithm in a search and rescue operation due to its completeness and optimality. In order to minimize the number of search states visited, the heuristic chosen is Manhattan distance. The implementation of A* search will not be discussed here since it's already been done in Section 4.1.4. For this part, the chosen

map is map 1. Since the previous test results show that Bob an goal positions are both reachable by the robot, map 1 is suitable for this part.

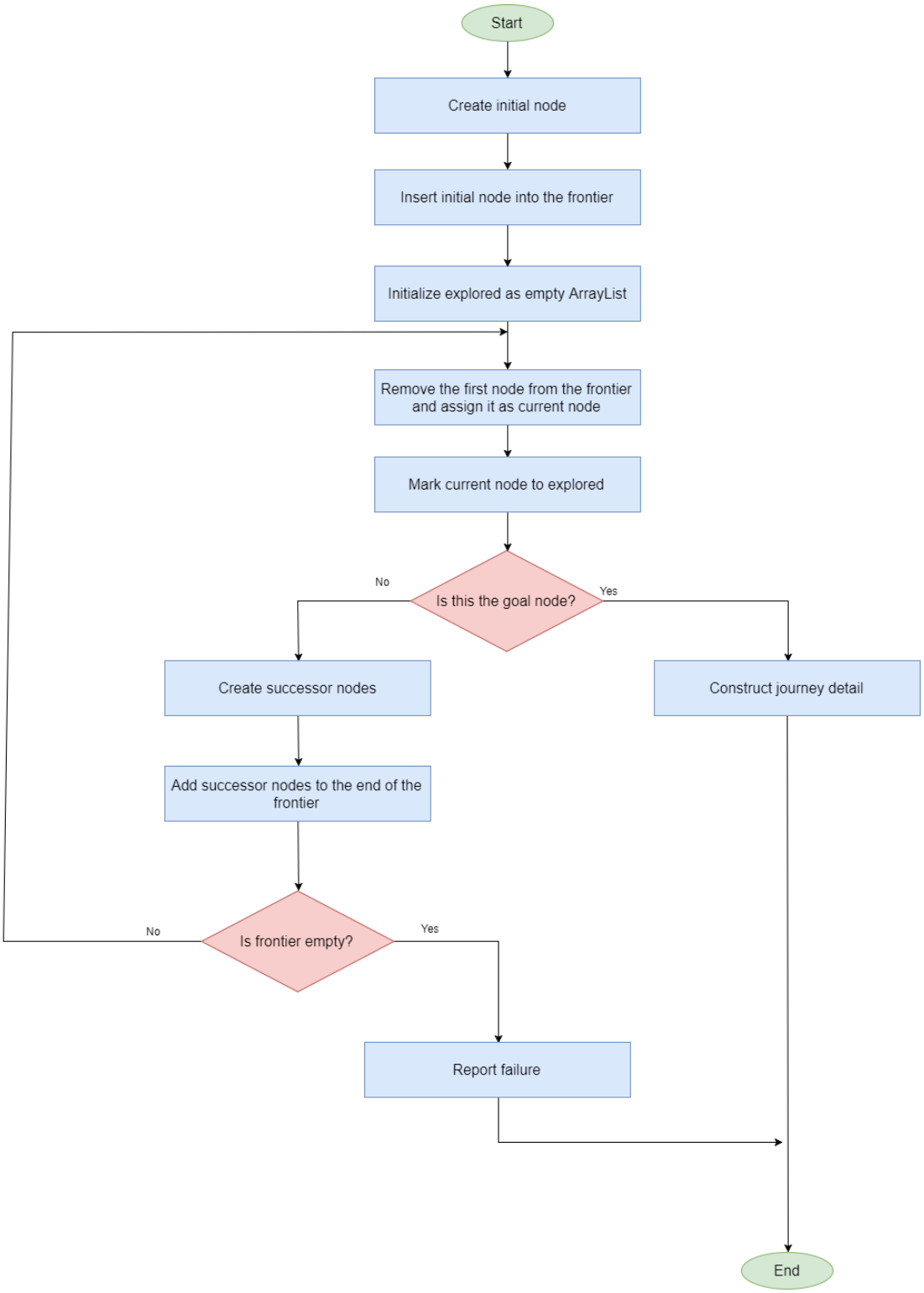

Figure 28: Flowchart showing the execution of the rescue operation for part 3.

Once the path to Bob and to goal is found, the program starts searching for sequence of actions that allows the robot to transport the survivors to safety. The flowchart of the

search is shown in Figure 28. The method ***getJourney()*** in ***RescueOperation*** class is used to perform search. The search starts with the state where the characters are at position B. Breadth First Search is used to find an optimal journey for this situation. BFS is chosen over DFS for this part because of its completeness and optimality. Due to time constraints, it is not possible to find out if there is any heuristic that can be used in this situation.

5.1.3 Journey Construction

```
1 private void constructPathToGoal(CharactersNode currentState) {
2     // construct path to goal by backtracking from goal to initial state
3     for(CharactersNode state = currentState; state != null; state = prev.get
    (state)) {
4           journey.add(state);
5       }
6     Collections.reverse(journey);
7   }
```

Listing 11: The code fragment used to construct the the journey detail.

As stated in the requirements, the system needs to be able to construct the evacuation detail. This was done by storing the mapping of each node to its parent node in the ***prev*** attribute of ***RescueOperation*** class (see Listing 11). Once the algorithm found the goal, it uses the code in Listing 11 to construct the evacuation detail by using the information stored in ***prev*** attribute to backtrack from the goal node to initial node, and reverse the order to get the final evacuation procedures.

5.2 Examples and Testing

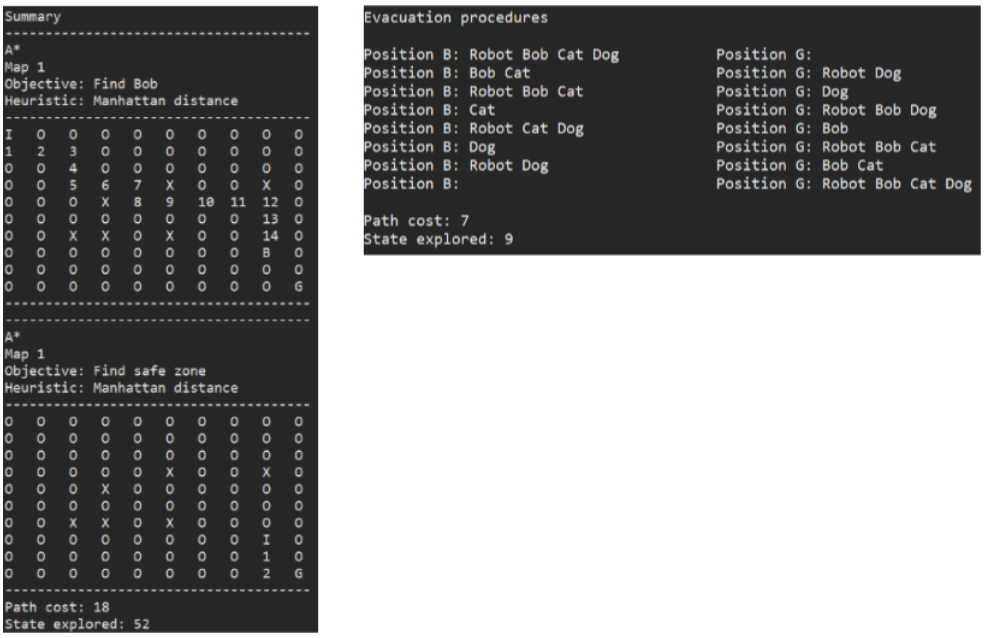

Figure 29: Printout from the execution of the program for part 3.

An example of part 3 program's execution is shown in Figure 29. On the left side, an optimal path to Bob and to goal position is shown. The right side shows the evacuation procedures of the robot. The evacuation procedures can be summarize as follows:

1. Robot and the survivors are at position B.
2. The robot takes the dog to position G.
3. The robot travels to position B alone.
4. The robot takes Bob to position G.
5. The robot brings the dog back to position B.
6. The robot brings the cat to position G.
7. The robot goes back to position B alone.
8. The robot brings the dog to position G.
9. Now everyone is at position G.

Algorithm	Search states visited	Path cost
Breadth First Search	9	7

Table 11: Performance summary for evacuation procedures

The summary of evacuation process's performance is shown in Table 11.

5.3 Evaluation

Overall, the implementation of part 3 managed to meet all the requirements states in system description. The program is able to an optimal path for the robot to fins Bob and the goal position. It also used BFS to find the sequence of actions that allows the robot to bring everyone to safety without ending up in an illegal state. Due to time constraints, it is not possible to find out about how can heuristic be incorporated into the evacuation problem. However, BFS did not perform badly in this situation. It manged to find the solution with very low number of states visited and path cost (9 and 7 respectively). This is because the there many constraints in this problem, which helps lower the number of successor nodes generated at each step. In short, the number of possible actions at each stage is relatively low.

6 Running

6.1 Search 1

6.1.1 Running Parameters

Algorithm	Parameter
Breadth First Search	BFS
Depth First Search	DFS

Table 12: Algorithm parameters for running Search1.jar.

Map	Parameter
Map 1	1
Map 2	2
Map 3	3
Map 4	4
Map 5	5
Map 6	6

Table 13: Map parameters for running Search1.jar.

6.1.2 Running Instructions

Search1.jar can be executed with the command:

```
java -jar Search1.jar algorithm map
```

Listing 12: The command used to run Search1.jar.

To run Search1.jar from AI-Search folder with BFS as search algorithm in map 1, the command in Listing 13 should be used.

```
java -jar Search1.jar BFS 1
```

Listing 13: The command used to run Search1.jar from AI-Search folder.

please consult parameters guide in Table 12 and Table 13 when running with different parameters.

6.2 Search 2

6.2.1 Running Parameters

Algorithm	Parameter
Best First Search	BestFS
A* Search	A*

Table 14: Algorithm parameters for running Search2.jar.

Heuristic	Parameter
Manhattan distance	M
Euclidean distance	E
Combination	all

Table 15: Heuristic parameters for running Search2.jar.

Map	Parameter
Map 1	1
Map 2	2
Map 3	3
Map 4	4
Map 5	5
Map 6	6

Table 16: Map parameters for running Search2.jar.

6.2.2 Running Instructions

Search2.jar can be executed with the command:

```
1 java -jar Search2.jar algorithm heuristic map
```

Listing 14: The command used to run Search2.jar.

To run Search2.jar from AI-Search folder with A* as search algorithm, and Manhattan distance as heuristic in map 1, the command in Listing 15 should be used.

```
java -jar Search2.jar A* M 1
```

Listing 15: The command used to run Search2.jar from AI-Search folder.

please consult parameters guide in Table 14, Table 15 and Table 16 when running with different parameters.

6.3 Search 3

6.3.1 Running Instructions

Search3.jar can be executed with the command:

```
java -jar Search3.jar
```

Listing 16: The command used to run Search3.jar.

Word count: 4397

Bibliography

GeeksforGeeks (2017), 'Applications of depth first search', http://www.geeksforgeeks.org/applications-of-depth-first-search/. Accessed: 2017-10-31.

Jain, N. (2017), 'Applications of breadth first traversal', http://www.geeksforgeeks.org/applications-of-breadth-first-traversal/. Accessed: 2017-10-30.

Kelly, K. & Labute, P. (1996), 'The a* search and applications to sequence alignment', https://www.chemcomp.com/journal/astar.htm. Accessed: 2017-10-31.

Mussmann, S. & See, A. (n.d.), 'Graph search algorithms', http://cs.stanford.edu/people/abisee/gs.pdf. Accessed: 2017-10-31.

Russell, S. J. & Norvig, P. (2009), *Artificial Intelligence: A Modern Approach*, 3 edn, Pearson Education.

Toniolo, A. (2017*a*), 'Cs5011: Assignment 2 - search - rescue simulations', https://studres.cs.st-andrews.ac.uk/CS5011/Practicals/A2/A2.pdf. Accessed: 2017-10-25.

Toniolo, A. (2017*b*), 'Cs5011 search part b', https://studres.cs.st-andrews.ac.uk/CS5011/Lectures/L9_w5.pdf. Accessed: 2017-10-25.