CS5011: Assignment 1 – Artificial Neural Networks

List of the implemented parts and extensions

Part 1: Search1.jar

1. Implementation of breadth-first search.
2. Implementation of depth-first search.

Part 2: Search2.jar

1. Implementation of best-first search.
2. Implementation of A* search.

Part 3: Search3_1.jar & Search3_3.jar

1. Implementation of bidirectional search.
2. Implementation of the four-robot team search.

Literature Review and Introduction

Searching is a universal problem-solving mechanism in artificial intelligence (Korf, 2010). Search algorithms can be classified into informed search algorithms and uninformed search algorithms. The uninformed algorithms, including breadth-first search depth-first search and bidirectional search, are given with only the definition of the problem without additional knowledge, (Russel and Norvig, 2005). For informed search algorithms, nodes are selected for expansion based on an evaluation function that estimates partial or total cost to the goal. Before the search is executed, the problem must be well defined with five parts (Russel and Norvig, 2005): the initial states (where the robot is initially), a set of actions (how the robot moves), a transition model (how the next state is generated), a goal test function (where to stop), a path cost function (how much each step costs), and a state space (all the possible locations of the robot). Russel and Norvig (2005) also proposed four criteria to measure and compare the performance of different search algorithms: completeness, optimality, time complexity and space complexity. In this assignment, breadth-first search, depth-first search, bidirectional search, bestfirst search with different heuristics and A* search are implemented and compared according to the four criteria.

Part 1 BFS and DFS

Design

1. Object design

A class named Node is designed according to the UML diagram (Fig. 1). For the instances of this class, the state is initialized by the constructor to save the x- and y- coordinates. The state space is determined by the map loaded by the program – all the possible positions on the map. The initial state is expressed by the root node instantiated by the coordinates of character to start (‘I’ or ‘B’ in this problem) on the given map. To test if the goal is reached, every time the tree is expanded, the system will check each frontier state if its location is identical to that of goal (‘B’ or ‘G’ in this problem). The states are stored in the ‘parent’ attribute of the frontier node – when the goal is reached by one of the frontier nodes, the parent attribute allows us to trace back to the origin along the tour, which gives the state when it is needed.

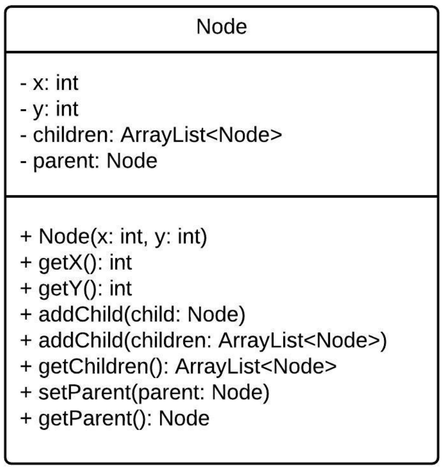


Figure 1. The UML diagram of Node.class for part 1.

In terms of the successor function, for each frontier node to be expanded, the order to add its adjacent clear cell to the frontier is south > east > north > west, though other successor functions are valid as well.

2. Implementation of searching strategies

   a) Depth-first

   For depth-first search (BFS), the search frontiers are stored in a last-in-first-out (LIFO) stack structure. After forming a frontier list and explored list with the initial state, the system will loop until the frontier list is empty or the goal is reached. For each searching step, the system executes the following procedures:

   i. Check if the frontier stack is empty – return an empty list to mark failure if empty.
   ii. Extract the node from the top of the stack.
   iii. Check if this node has any unexplored adjacent cells, remove it from the stack if it doesn't.
   iv. Expand the tree with a node initialized by the first adjacent grid (the priority is set by the successor function).
   v. Put this child node to the top of the stack and to the explored node list.
   vi. Check if the goal is reached – generate and return the state sequence of route if reached.

   b) Breadth-first

   For breadth-first search (DFS), the search frontiers are stored in a first-in-first-out (FIFO) queue structure. I did a slight optimization with BFS algorithm allowing the system to expand all of the frontier nodes simultaneously, and override the addChild() method of the node class with an ArrayList as input. The order of the queue is preserved by the successor function (Fig. 2). The frontier and explored lists are initialized in the same manner as DFS. For each step, the system executes the following procedures:

   i. Check if the frontier queue is empty – return an empty list to mark failure if empty.

ii. Extract the all nodes from the frontier queue.
iii. Find all of their unexplored adjacent cells as their children.
iv. Add these lists of children nodes to the tail of the frontier queue.
v. Remove the expanded parent nodes from the head of the queue.
vi. Check if the goal is reached – generate and return the state sequence of route if reached.

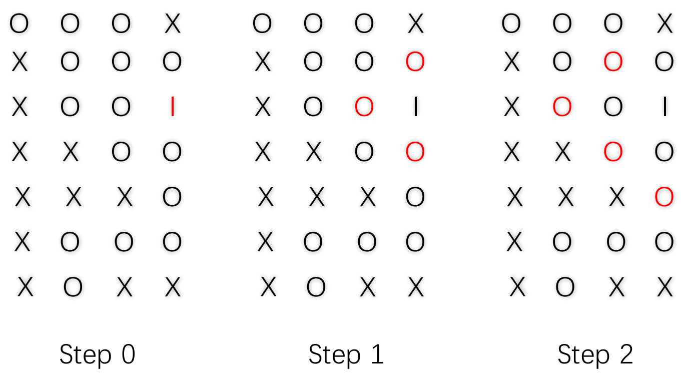


Figure 2. The map is a portion of map6. The locations frontier nodes are marked red. For each step, all of the frontier nodes are expanded and then removed from the frontier list.

c) Comparing the implementations

The first and last part of BFS and DFS is ultimately the same. The difference between the two implementations is caused by the data structures we choose to store the frontier nodes. In this assignment, 'ArrayList' is modified to satisfy both data structure.

In traditional BFS, we always expand and examine the 'oldest' node first. Therefore, we only retrieve the first element from the ArrayList and add new elements to its rear. In this assignment, all of the frontiers are expanded and examined in a single step to accomplish the same goal for this problem.

In DFS, the 'newer' nodes are expanded and examined before the 'older' ones. In other words, we also only retrieve the first element from the ArrayList but add new elements to its head.

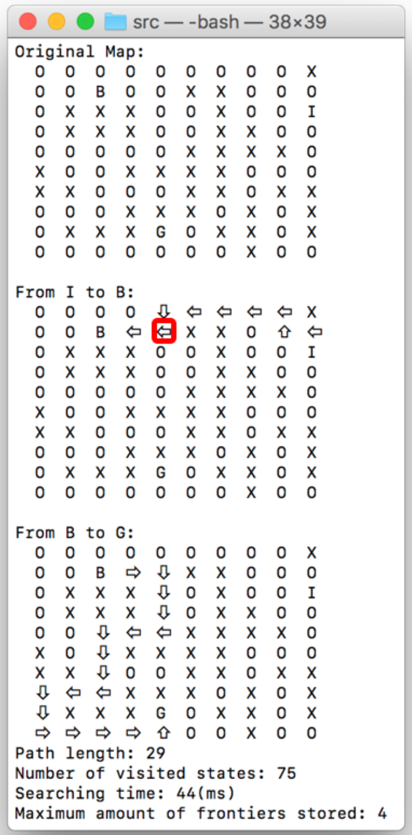
3. System output

During each step of searching, the coordinates of all current frontier nodes are printed. If the system successfully finds a valid solution from ‘I’ to ‘B’ and ‘B’ to ‘G’, the route map will be printed as well as some statistics illustrating how good the solution is, space efficiency and time efficiency. The two routes are printed separately with arrows because they may sometimes overlap.

Examples and Testing

Figure 3a and 3b show two examples for BFS and DFS tested on the map6. Both methods managed to find path from ‘I’ to ‘B’ and ‘B’ to ‘G’ on map6. The frontier coordinates can be found when running the program.

a)

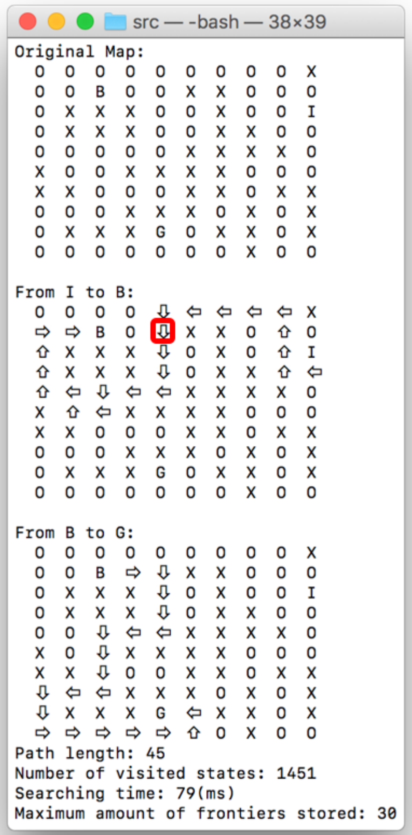


```
Original Map:
  O  O  O  O  O  O  O  O  O  X
  O  O  B  O  O  X  X  O  O  O
  O  X  X  X  O  O  X  O  O  I
  O  X  X  X  O  O  X  X  O  O
  O  O  O  O  O  X  X  X  X  O
  X  O  O  X  X  X  X  O  O  O
  X  X  O  O  O  X  X  O  X  X
  O  O  O  X  X  X  O  X  O  X
  O  X  X  X  G  O  X  X  O  X
  O  O  O  O  O  O  O  X  O  O

From I to B:
  O  O  O  O  ⇩  ⇦  ⇦  ⇦  ⇦  X
  O  O  B  ⇦  ⇦  X  X  O  ⇧  ⇦
  O  X  X  X  O  O  X  O  O  I
  O  X  X  X  O  O  X  X  O  O
  O  O  O  O  O  X  X  X  X  O
  X  O  O  X  X  X  X  O  O  O
  X  X  O  O  O  X  X  O  X  X
  O  O  O  X  X  X  O  X  O  X
  O  X  X  X  G  O  X  X  O  X
  O  O  O  O  O  O  O  X  O  O

From B to G:
  O  O  O  O  O  O  O  O  O  X
  O  O  B  ⇨  ⇩  X  X  O  O  O
  O  X  X  X  ⇩  O  X  O  O  I
  O  X  X  X  ⇩  O  X  X  O  O
  O  O  ⇩  ⇦  ⇦  X  X  X  X  O
  X  O  ⇩  X  X  X  X  O  O  O
  X  X  ⇩  O  O  X  X  O  X  X
  ⇩  ⇦  ⇦  X  X  X  O  X  O  X
  ⇩  X  X  X  G  O  X  X  O  X
  ⇨  ⇨  ⇨  ⇨  ⇧  O  O  X  O  O
Path length: 29
Number of visited states: 75
Searching time: 44(ms)
Maximum amount of frontiers stored: 4
```

b)



```
Original Map:
  O  O  O  O  O  O  O  O  O  X
  O  O  B  O  O  X  X  O  O  O
  O  X  X  X  O  O  X  O  O  I
  O  X  X  X  O  O  X  X  O  O
  O  O  O  O  O  X  X  X  X  O
  X  O  O  X  X  X  X  O  O  O
  X  X  O  O  O  X  X  O  X  X
  O  O  O  X  X  X  O  X  O  X
  O  X  X  X  G  O  X  X  O  X
  O  O  O  O  O  O  O  X  O  O

From I to B:
  O  O  O  O  ⇩  ⇦  ⇦  ⇦  ⇦  X
  ⇨  ⇨  B  O  ⇩  X  X  O  ⇧  O
  ⇧  X  X  X  ⇩  O  X  O  ⇧  I
  ⇧  X  X  X  ⇩  O  X  X  ⇧  ⇦
  ⇧  ⇦  ⇩  ⇦  ⇦  X  X  X  X  O
  X  ⇧  ⇦  X  X  X  X  O  O  O
  X  X  O  O  O  X  X  O  X  X
  O  O  O  X  X  X  O  X  O  X
  O  X  X  X  G  O  X  X  O  X
  O  O  O  O  O  O  O  X  O  O

From B to G:
  O  O  O  O  O  O  O  O  O  X
  O  O  B  ⇨  ⇩  X  X  O  O  O
  O  X  X  X  ⇩  O  X  O  O  I
  O  X  X  X  ⇩  O  X  X  O  O
  O  O  ⇩  ⇦  ⇦  X  X  X  X  O
  X  O  ⇩  X  X  X  X  O  O  O
  X  X  ⇩  O  O  X  X  O  X  X
  ⇩  ⇦  ⇦  X  X  X  O  X  O  X
  ⇩  X  X  X  G  ⇦  X  X  O  X
  ⇨  ⇨  ⇨  ⇨  ⇨  ⇧  O  X  O  O
Path length: 45
Number of visited states: 1451
Searching time: 79(ms)
Maximum amount of frontiers stored: 30
```

Figure 3. Original map, two route maps and some running statistics for (a) BFS and (b) DFS on map6. The red boxes mark a different searching decision between BFS and DFS.

Evaluation

To evaluate the performances of BFS and DFS for this problem with provided data. The route lengths computed by BFS and DFS on provided maps are shown in table 1. Generally speaking, BFS gives better (shorter) routes than DFS. Note that there is no possible route on map4 and map5.

Table 1. Route length solutions on provided map given by BFS and DFS.

Map	**map1**	**map2**	**map3**	**map4**	**map5**	**map6**
BFS	18	34	21	N/A	N/A	29
DFS	42	54	59	N/A	N/A	45

Several other attributes are also considered to compare the time and space efficiencies of BFS and DFS (Table 2). BFS demonstrates much better performance in terms of all the considered attributes. Amount of maximum stored frontiers indicates the space efficiency; number of visited states and searching time indicate the time efficiency.

Table 2. Evaluation of BFS and DFS with performance indicators. These data are given by running on the lab computer (Core i5-4400 CPU @ 3.10GHz).

Attributes	Number of visited states					Searching time (ms)					Maximum stored frontiers				
Map	map1	map2	map3	map6	avg.	map1	map2	map3	map6	avg.	map1	map2	map3	map6	avg.
BFS	96	145	120	75	109	5	8	6	8	6.75	9	7	9	4	7.25
DFS	830	4602	1150	1451	2008	12	26	14	17	17.25	40	53	33	30	39

To compare the two algorithms, a route divergence is marked by the red boxes in figure 3. In DFS, the nodes always choose its successor according to the successor function. Therefore, it often ends up taking a detour. On the contrary, BFS guarantees an optimal solution (shortest route) in this problem. For each searching step, all of the frontiers simultaneously expand one cell further and all of the possible goals within the same path cost is represented by the frontiers in the current state. Therefore, once one of the frontiers reaches the destination, its route is guaranteed to be the shortest route in this case.

Running

Part 1 includes one jar file.

Both DFS and BFS algorithm is implemented in Search1.jar, and all six maps is stored. To run the program, use

“java –jar Search1.jar [map] [strategy]”.

Valid map attributes include “map1”, “map2”, “map3”, “map4”, “map5”, and “map6”.

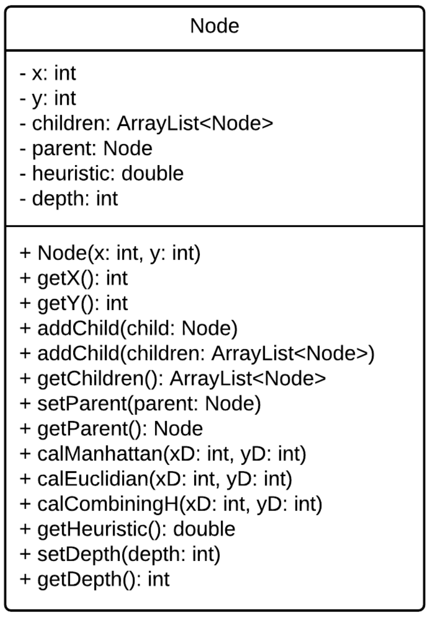
Valid strategy attributes include “bfs” and “dfs”.

Part 2 BestFS and A*

Design

1. Object design

The Node class is similar to the class with the same name in part 1 with two additional attributes: heuristic and depth, and relevant methods (Fig. 4). The addChild() method is also modified to set the children’s depth to be parent’s depth+1 to represent current path cost. The heuristic attribute represents the estimated cost of the path from the state at this node to the goal. To calculate the Manhattan distance, Euclidian distance or the combined heuristic, the coordinates of the destination is passed to invoke the corresponding methods. Other features are extended from the Node class in part 1.



```
Node
- x: int
- y: int
- children: ArrayList<Node>
- parent: Node
- heuristic: double
- depth: int
+ Node(x: int, y: int)
+ getX(): int
+ getY(): int
+ addChild(child: Node)
+ addChild(children: ArrayList<Node>)
+ getChildren(): ArrayList<Node>
+ setParent(parent: Node)
+ getParent(): Node
+ calManhattan(xD: int, yD: int)
+ calEuclidian(xD: int, yD: int)
+ calCombiningH(xD: int, yD: int)
+ getHeuristic(): double
+ setDepth(depth: int)
+ getDepth(): int
```

Figure 4. The UML diagram of Node.class for part 2.

2. Implementation of searching strategies

a) Best-first Search (BestFS)

The fundamental issue for BestFS is to find a proper heuristic to optimize the choice of expanded node. In this distance-based problem, Euclidian Distance, Manhattan Distance and Chebyshev Distance (Fig. 5) are equally valid. In this assignment, Manhattan distance, Euclidian distance and a combination of the two (by finding the dominating one) heuristics are implemented to compare their performances. All of the states are stored in a priority queue, implemented by Array List as well. After forming a frontier list and explored list with the initial state, the system will loop until the frontier list is empty or the goal is reached. For each searching step, the system executes following procedures:

i. Check if the frontier queue is empty – return an empty list to mark failure if empty.
ii. Retrieve the first state from the frontier priority-queue.
iii. Find all of their unexplored adjacent cells as their children.
iv. Calculate the heuristic of the children nodes.

v. Insert each child to the priority queue. The rank is determined by the heuristic in an increasing order.

vi. Remove the expanded parent nodes from the head of the queue.

vii. Check if the goal is reached – generate and return the state sequence of route if reached.

b) A* Search

A* search also use priority queue to store all the states, however, the priority is a combination of the cost to get to the current node and the distance-based heuristics adopted in the BestFS. For each searching step, the system follows the same procedures of BestFS, except for v. The rank is now determined by the f(n), given by

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

where $g(n)$ is the cost of path from the start to this node, and $h(n)$ is the Manhattan distance from this node to the goal.

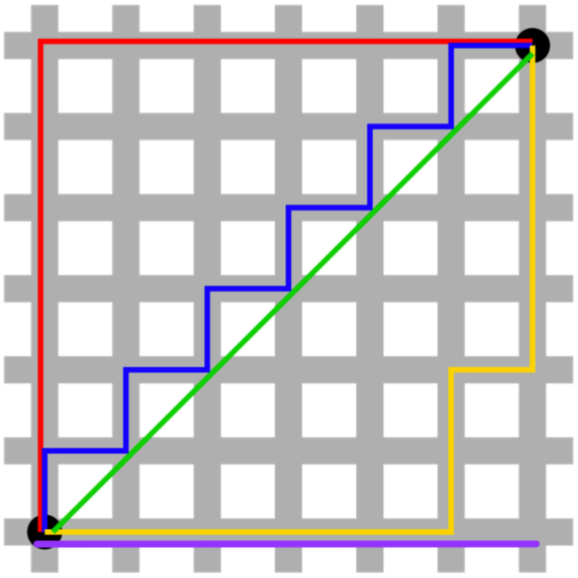


Figure 5. Distance-based heuristics: Manhattan distance (red, blue, and yellow lines), Euclidian distance (green line), and Chebyshev distance (purple line) (From the lecture slide).

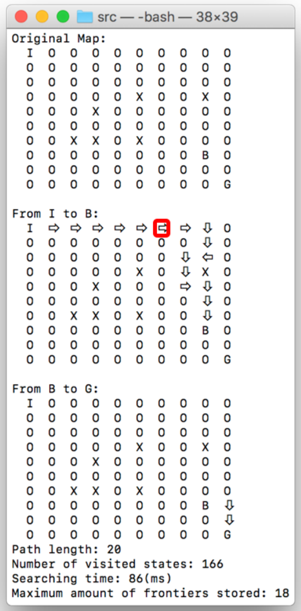
3. System output

The output of part 2 is the same as part 1, with all the frontiers printed for each state and solutions.

Examples and Testing

Figure 6a, 6b and 6c show the solutions on map1 given by BestFS (Manhattan), BestFS (Euclidian) and A* (Manhattan). All three algorithms managed to give correct solutions.

a)

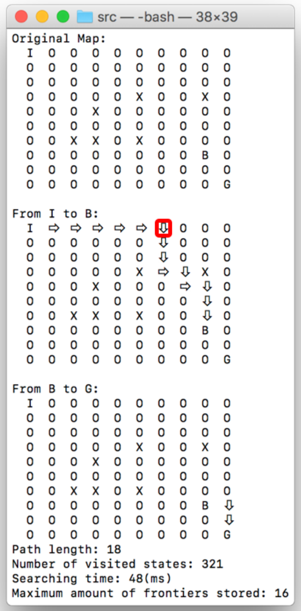


```
src — -bash — 38×39
Original Map:
  I  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  X  O  O  X  O
  O  O  O  X  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  X  X  O  X  O  O  O  O
  O  O  O  O  O  O  O  O  B  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  G

From I to B:
  I  ⇨  ⇨  ⇨  ⇨  ⇨  ⇨  ⇨  ⇩  O
  O  O  O  O  O  O  O  O  ⇩  O
  O  O  O  O  O  O  O  ⇩  ⇦  O
  O  O  O  O  O  X  O  ⇩  X  O
  O  O  O  X  O  O  O  ⇨  ⇩  O
  O  O  O  O  O  O  O  O  ⇩  O
  O  O  X  X  O  X  O  O  ⇩  O
  O  O  O  O  O  O  O  O  B  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  G

From B to G:
  I  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  X  O  O  X  O
  O  O  O  X  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  X  X  O  X  O  O  O  O
  O  O  O  O  O  O  O  O  B  ⇩
  O  O  O  O  O  O  O  O  O  ⇩
  O  O  O  O  O  O  O  O  O  G
Path length: 20
Number of visited states: 166
Searching time: 86(ms)
Maximum amount of frontiers stored: 18
```

b)

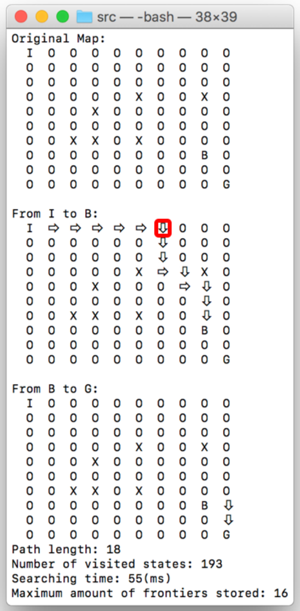


```
src — -bash — 38×39
Original Map:
  I  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  X  O  O  X  O
  O  O  O  X  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  X  X  O  X  O  O  O  O
  O  O  O  O  O  O  O  O  B  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  G

From I to B:
  I  ⇨  ⇨  ⇨  ⇨  ⇨  ⇩  O  O  O
  O  O  O  O  O  O  ⇩  O  O  O
  O  O  O  O  O  O  ⇩  O  O  O
  O  O  O  O  O  X  ⇨  ⇩  X  O
  O  O  O  X  O  O  O  ⇨  ⇩  O
  O  O  O  O  O  O  O  O  ⇩  O
  O  O  X  X  O  X  O  O  ⇩  O
  O  O  O  O  O  O  O  O  B  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  G

From B to G:
  I  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  X  O  O  X  O
  O  O  O  X  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  X  X  O  X  O  O  O  O
  O  O  O  O  O  O  O  O  B  ⇩
  O  O  O  O  O  O  O  O  O  ⇩
  O  O  O  O  O  O  O  O  O  G
Path length: 18
Number of visited states: 321
Searching time: 48(ms)
Maximum amount of frontiers stored: 16
```

c)



```
src — -bash — 38×39
Original Map:
  I  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  X  O  O  X  O
  O  O  O  X  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  X  X  O  X  O  O  O  O
  O  O  O  O  O  O  O  O  B  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  G

From I to B:
  I  ⇨  ⇨  ⇨  ⇨  ⇨  ⇩  O  O  O
  O  O  O  O  O  O  ⇩  O  O  O
  O  O  O  O  O  O  ⇩  O  O  O
  O  O  O  O  O  X  ⇨  ⇩  X  O
  O  O  O  X  O  O  O  ⇨  ⇩  O
  O  O  O  O  O  O  O  O  ⇩  O
  O  O  X  X  O  X  O  O  ⇩  O
  O  O  O  O  O  O  O  O  B  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  G

From B to G:
  I  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  X  O  O  X  O
  O  O  O  X  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  X  X  O  X  O  O  O  O
  O  O  O  O  O  O  O  O  B  ⇩
  O  O  O  O  O  O  O  O  O  ⇩
  O  O  O  O  O  O  O  O  O  G
Path length: 18
Number of visited states: 193
Searching time: 55(ms)
Maximum amount of frontiers stored: 16
```

Figure 6. Original map, two route maps and some running statistics for (a) BestFS with Manhattan, (b) BestFS with Euclidian and (c) A* with Manhattan on map1. The red boxes mark a different searching decision. The map is the provided named “map1”.

Evaluation

The A* algorithm guarantees to give the optimal solution (shortest path) by design. Tested by all six maps, BestFS algorithms only failed to find the shortest path on map1 when it's using Manhattan distance (Table 3). In this grid searching problem, Manhattan distance always dominates the other, so the BestFS with combining heuristic is essentially the same as BestFS (M).

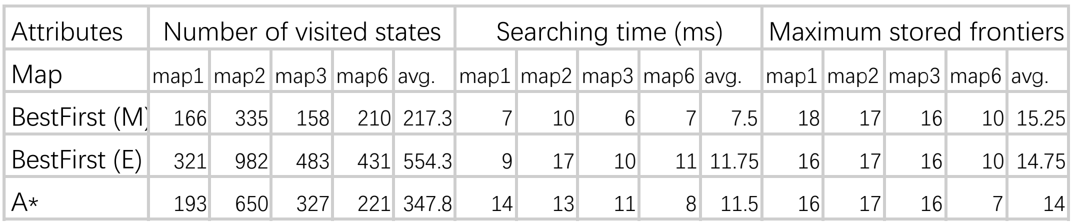
Table 3. Route length solutions on provided map given by BestFS and A*.

Map	**map1**	**map2**	**map3**	**map4**	**map5**	**map6**
BestFS (M)	20	34	21	N/A	N/A	29
BestFS (E)	18	34	21	N/A	N/A	29
A*	18	34	21	N/A	N/A	29

The space and time efficiencies are indicated by the statistics in the following table. Though BestFS(M) shows weaker accuracy (75%) for optimal searching, it presents best time efficiency among the three algorithms with least visited states and shortest searching time, while A* shows better space efficiency. Since the Manhattan distance methods shows better efficiency and A* is an optimal algorithm by design, the A* implement Manhattan distance to calculate $h(n)$ in equation (1).

To better evaluate the performances of these three algorithms, more testing data are expected to be included. BestFS and A* will be compared to BFS, DFS and bidirectional search in the following section.

Table 4. Evaluation of BestFS and A* with performance indicators. These data are given by running on the lab computer (Core i5-4400 CPU @ 3.10GHz).



Attributes	Number of visited states					Searching time (ms)					Maximum stored frontiers				
Map	map1	map2	map3	map6	avg.	map1	map2	map3	map6	avg.	map1	map2	map3	map6	avg.
BestFirst (M)	166	335	158	210	217.3	7	10	6	7	7.5	18	17	16	10	15.25
BestFirst (E)	321	982	483	431	554.3	9	17	10	11	11.75	16	17	16	10	14.75
A*	193	650	327	221	347.8	14	13	11	8	11.5	16	17	16	7	14

Running

Part 2 includes one jar file.

Both BestFS (with Manhattan distance or Euclidian distance) and A* algorithm is implemented in Search2.jar, and all six maps is stored. To run the program, use

“java –jar Search2.jar [map] [strategy]”.

Valid map attributes include “map1”, “map2”, “map3”, “map4”, “map5”, and “map6”.

Valid strategy attributes include “BestFirstM”, “BestFirstE”, “BestFirstCombine” and “A*”.

Part 3.1 Bidirectional Search

Design

1. Object design

Same as part 1.

2. Implementation of searching strategies

The bidirectional essentially runs two searches simultaneously: a forward one searching from the origin and a backward one searching from the goal. The searching stops when the two meet in the middle. In this part of assignment, BFS is implemented for both forward and backward search.

The workflow for bidirectional search is listed as followed:

i. Check if either the forward-searching frontier queue or backward-searching frontier is empty – return an empty list to mark failure if empty.

ii. Expand the forward tree and remove the frontiers expanded in the last step from the queue.

iii. Expand the backward tree and remove the frontiers expanded in the last step from the queue.

iv. Use a two-layer nested loop to check if the two trees meet in the middle. Generate and return results if meet.

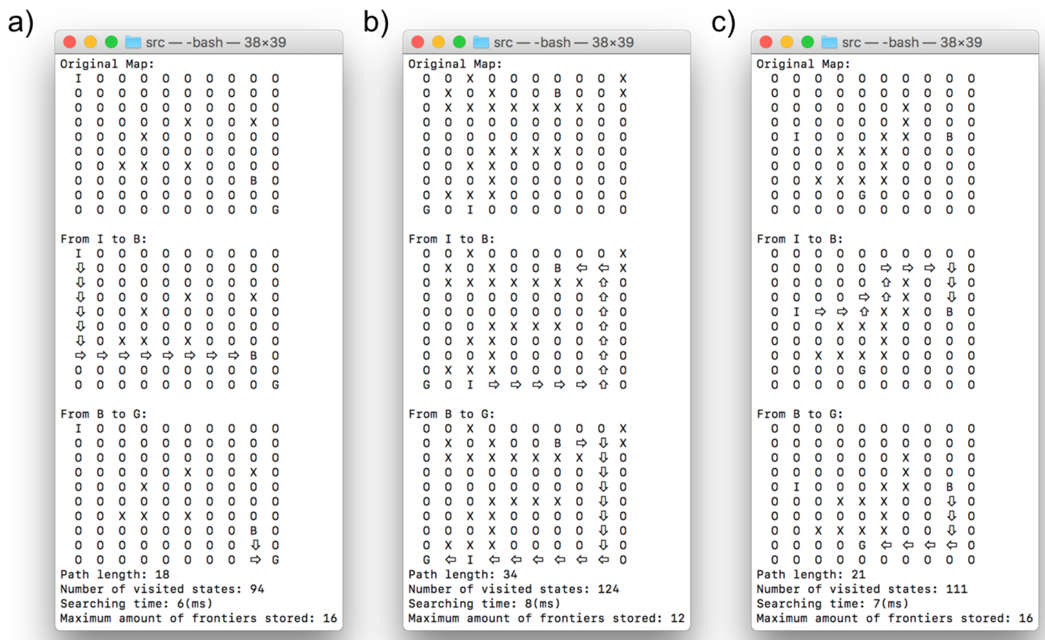
It is worth noticing that in this workflow, the we only remove the frontiers that are expanded in the former step, instead of removing nodes immediately after expanded as in part 1 and part 2. The reason for doing this is that the frontiers from two ends may miss each other when they both go one step ahead. An alternative approach is to invoke another goal-checking between step ii. and step iii. The advantage of the chosen approach is better time efficiency with doubled memory usage, and the opposite for this alternative method.

3. System output

The output for this part is similar to part 1 and part 2, with printed frontiers from both ends.

Examples and Testing

Figure 7a, 7b and 7c show the solutions on map1, map2 and map3 respectively, given by the bidirectional search algorithm. All of these solutions are optimal.



a)

```
Original Map:
  I  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  X  O  O  X  O
  O  O  O  X  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  X  X  O  X  O  O  O  O
  O  O  O  O  O  O  O  O  B  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  G

From I to B:
  I  O  O  O  O  O  O  O  O  O
  ⇩  O  O  O  O  O  O  O  O  O
  ⇩  O  O  O  O  O  O  O  O  O
  ⇩  O  O  O  O  X  O  O  X  O
  ⇩  O  O  X  O  O  O  O  O  O
  ⇩  O  O  O  O  O  O  O  O  O
  ⇩  O  X  X  O  X  O  O  O  O
  ⇨  ⇨  ⇨  ⇨  ⇨  ⇨  ⇨  ⇨  B  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  G

From B to G:
  I  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  X  O  O  X  O
  O  O  O  X  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  X  X  O  X  O  O  O  O
  O  O  O  O  O  O  O  O  B  O
  O  O  O  O  O  O  O  O  ⇩  O
  O  O  O  O  O  O  O  O  ⇨  G
Path length: 18
Number of visited states: 94
Searching time: 6(ms)
Maximum amount of frontiers stored: 16
```

b)

```
Original Map:
  O  O  X  O  O  O  O  O  O  X
  O  X  O  X  O  O  B  O  O  X
  O  X  X  X  X  X  X  X  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  X  X  X  X  O  O  O
  O  O  X  X  O  O  O  O  O  O
  O  O  O  X  O  O  O  O  O  O
  O  X  X  X  O  O  O  O  O  O
  G  O  I  O  O  O  O  O  O  O

From I to B:
  O  O  X  O  O  O  O  O  O  X
  O  X  O  X  O  O  B  ⇦  ⇦  X
  O  X  X  X  X  X  X  X  ⇧  O
  O  O  O  O  O  O  O  O  ⇧  O
  O  O  O  O  O  O  O  O  ⇧  O
  O  O  O  X  X  X  X  O  ⇧  O
  O  O  X  X  O  O  O  O  ⇧  O
  O  O  O  X  O  O  O  O  ⇧  O
  O  X  X  X  O  O  O  O  ⇧  O
  G  O  I  ⇨  ⇨  ⇨  ⇨  ⇨  ⇧  O

From B to G:
  O  O  X  O  O  O  O  O  O  X
  O  X  O  X  O  O  B  ⇨  ⇩  X
  O  X  X  X  X  X  X  X  ⇩  O
  O  O  O  O  O  O  O  O  ⇩  O
  O  O  O  O  O  O  O  O  ⇩  O
  O  O  O  X  X  X  X  O  ⇩  O
  O  O  X  X  O  O  O  O  ⇩  O
  O  O  O  X  O  O  O  O  ⇩  O
  O  X  X  X  O  O  O  O  ⇩  O
  G  ⇦  I  ⇦  ⇦  ⇦  ⇦  ⇦  ⇦  O
Path length: 34
Number of visited states: 124
Searching time: 8(ms)
Maximum amount of frontiers stored: 12
```

c)

```
Original Map:
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  X  O  O  O
  O  O  O  O  O  O  X  O  O  O
  O  I  O  O  O  X  X  O  B  O
  O  O  O  X  X  X  O  O  O  O
  O  O  O  O  O  X  O  O  O  O
  O  O  X  X  X  X  O  O  O  O
  O  O  O  O  G  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O

From I to B:
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  ⇨  ⇨  ⇨  ⇩  O
  O  O  O  O  O  ⇧  X  O  ⇩  O
  O  O  O  O  ⇨  ⇧  X  O  ⇩  O
  O  I  ⇨  ⇨  ⇧  X  X  O  B  O
  O  O  O  X  X  X  O  O  O  O
  O  O  O  O  O  X  O  O  O  O
  O  O  X  X  X  X  O  O  O  O
  O  O  O  O  G  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O

From B to G:
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  O  O  O  O
  O  O  O  O  O  O  X  O  O  O
  O  O  O  O  O  O  X  O  O  O
  O  I  O  O  O  X  X  O  B  O
  O  O  O  X  X  X  O  O  ⇩  O
  O  O  O  O  O  X  O  O  ⇩  O
  O  O  X  X  X  X  O  O  ⇩  O
  O  O  O  O  G  ⇦  ⇦  ⇦  ⇦  O
  O  O  O  O  O  O  O  O  O  O
Path length: 21
Number of visited states: 111
Searching time: 7(ms)
Maximum amount of frontiers stored: 16
```

Figure 7. Original map, two route maps and some running statistics on (a) map1, (b) map2 and

(c) map3. The implemented algorithm is bidirectional search.

Evaluation

In this part, all five searching algorithms are compared to evaluate their performance. As is mentioned in the literature review, four criteria are widely applied as performance indicators: completeness, optimality, time complexity and space complexity. It is worth noticing that the comparison and performance evaluation for these methods are specified for the route-searching problem in a 10×10 grid map. Although BFS has exponential space complexity and DFS has linear complexity, because BFS takes much less step to the goal, it demonstrates better space efficiency in this problem. The following evaluation may not apply when solving other search problems. All of the implemented algorithms fulfilled completeness, while BFS, bidirectional search and A* gives optimal solutions (shortest possible path) (Table 5). Besides, number of visited states, searching time and the amount of maximum stored frontiers are measured for each algorithm to demonstrate the time complexity and space complexity (Table 6 and Fig. 8). BFS visits the least states and requires shortest computing time. Although DFS visited almost ten times more states than the others, the searching times DFS is two to three times of the rest's, mainly due to simpler decision-making strategy of it. BestFS, A* and bidirectional search present similar space efficiencies considering the maximum temporal states storage, while BFS occupies the least storage as a result of the optimization implemented in part 1. To quantitatively evaluate their performances, I propose a marking strategy, given by

$$mark = time \times storage \times \frac{step_{solution}}{step_{optimal}} \tag{2}$$

where all the attributes are averaged from data from map1, map2, map3 and map6 (Table 7). Lower score indicates better performance. As is shown in table 6, BFS is the best algorithm overall for this problem. And the rest scores are normalized according to BFS, given by

$$normalized\ mark = \frac{mark_{BFS}}{mark} \times 100 \tag{3}$$

According to the mark, BFS ranks the top, followed by bidirectional search, BestFS(M), A*, and DFS in decreasing order. These marks are only based on the provided map data. To better evaluate this issue, more data should be considered. Future work is expected to optimize the bidirectional search with other basic searching strategy, like A*. Also, the statistics should also take failed maps into account.

Table 5. Completeness and optimality of the five algorithms for grid route-searching problem.

	BFS	DFS	BestFS	A*	Bidirectional
Completeness	√	√	√	√	√
Optimality	√	×	×	√	√

Table 6. Averaged performance indicators averaged over all four maps that have valid path. These data are given by running on the lab computer (Core i5-4400 CPU @ 3.10GHz).

Algorithm	Visited states	Time (ms)	Max. storage	Optimization
BFS	109	6.75	7.25	100%
DFS	2008	17.3	39	0%
BestFS (M)	217	7.5	15.3	75%
A*	348	11.5	14	100%
Bidirectional	101	7.25	13	100%

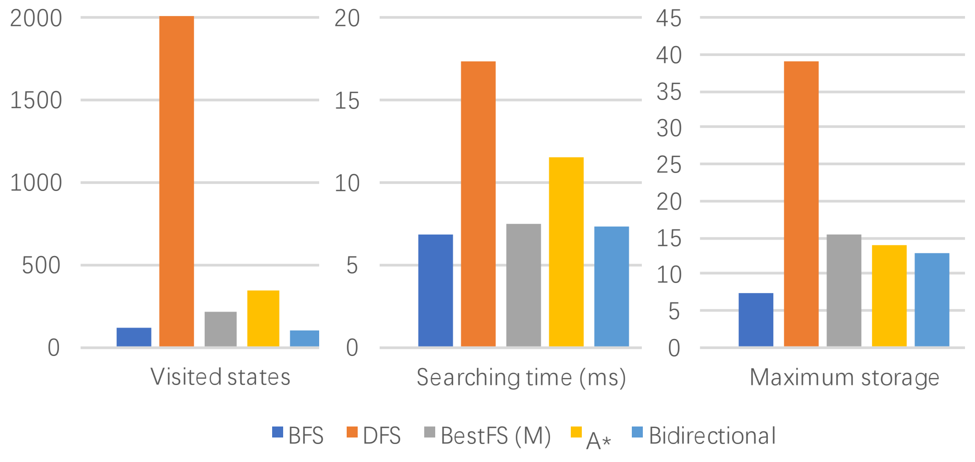


Figure 8. Bar charts of data in table 5.

Table 7. Parameters in function (2). The marks are normalized to the BFS (best algorithms for this problem) Two decimals kept in this table.

Algorithm	Time (ms)	Storage	Solution	Ratio	Mark	Normalized
BFS	6.75	7.25	25.50	1.00	48.94	100.00
DFS	17.30	39.00	50.00	1.96	1322.94	3.70
BestFS (M)	7.50	15.30	26.00	1.02	117.00	41.83
A*	11.50	14.00	25.50	1.00	161.00	30.40
Bidirectional	7.25	13.00	25.50	1.00	94.25	51.92

Running

Part 3.1 includes one jar file.

Bidirectional algorithm is implemented in Search3_1.jar, and all six maps is stored. To run the program, use

“java –jar Search3_1.jar [map]”.

Valid map attributes include “map1”, “map2”, “map3”, “map4”, “map5”, and “map6”.

Part 3.3 The Four-robot problem

Design

1. Object design
   Same as part 1.

2. Implementation of searching strategies
   BFS strategy is implemented to solve this search problem. A list of nodes ‘state’ is used to store a combination of the configuration. And the ‘frontier’ ArrayList contains a series of ‘state’. A problematic issue needs to be

tackled before searching is how to check whether the four robots are together. Though this can be done with dozens of ‘if’ sentences, I propose a simpler approach to count the sum of neighbor robots (Fig. 9). The minimum summation of neighbor robots is six to satisfy configurations where all robots are together. Therefore, a four-layer nested loop is used to check if the configuration is valid by comparing the summation with the minimum request, six.

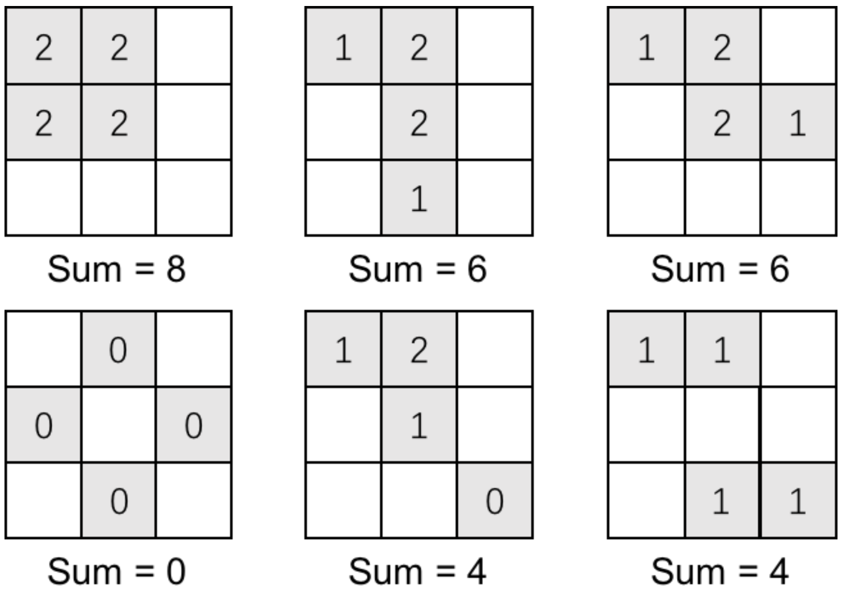


Figure 9. Examples of four-robot configurations. White cells are unoccupied; each gray cell contains one robot. The numbers inside count the amount of neighbor robots. And the sum of the counts lies behind each configuration. The first row contains three valid configurations; the lower three are invalid.

Another method is implemented to check if the current configuration reaches the goal to allow robots ending up with different relative location. The initial state is given by the locations of four ‘I’s on the map. The frontier queue and explored list are also initialized with the initial state as the first element. For each searching step, the workflow is:

i. Check if the frontier list is empty. Return an empty list to mark failure if empty.
ii. Retrieve all of the states stored in the frontier queue.
iii. For each state, expand its four nodes to their adjacent cell to

get their children nodes.

iv. Use a four-layer nested loop to check the validity of the configuration given by four of the child nodes.

v. Save valid and unexplored configurations as new states into the frontier queue and explored list.

vi. Remove all the expanded states from the frontier queue.

vii. Check if the goal is arrived. Generate and return the results.

3. System output

After the route is found, for each step, the configuration of the four robots are printed in the map. All robots are marked by a number.

Examples and Testing

For the provided map, it takes ten steps from the initial positions to victims and eight more steps from the victims to the safe places. The first ten steps are show below (Fig. 10). The program will produce the entire route.

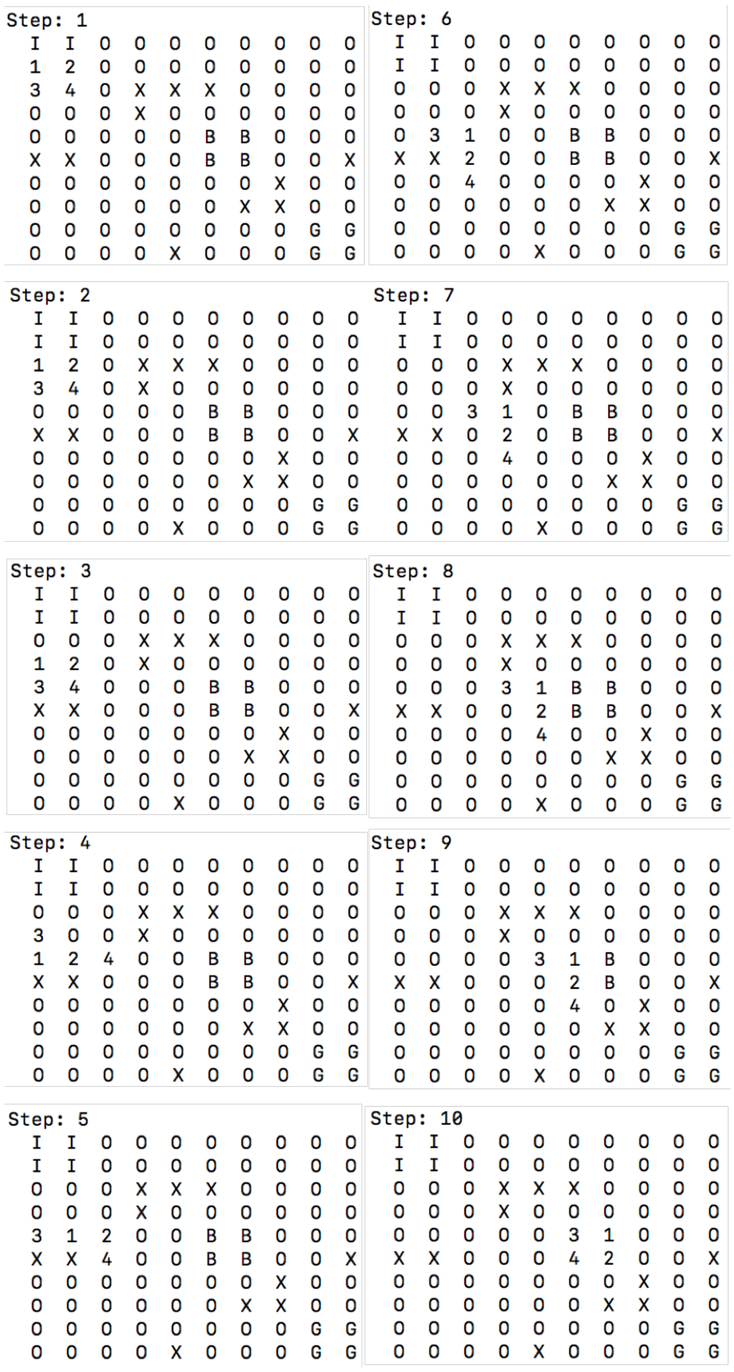
Evaluation

Though the program gives the correct answer and the answer should be optimal due to advantages of BFS in this problem. However, it takes notably long time to run. It can be improved by giving a proper heuristic to reduce the examined states. The program is written in a manner to adapt to other maps symbolized by ‘O’, ‘X’, ‘I’, ‘B’ and ‘G’. An enhanced version should allow it to adapt to group-of-five or group-of-three robot team.

Running

Part 3.3 includes one jar file.

Bidirectional algorithm is implemented in Search3_1.jar, only one map is stored. To run the program, use “java –jar Search3_1.jar”.



```
Step: 1
 I  I  O  O  O  O  O  O  O  O
 1  2  O  O  O  O  O  O  O  O
 3  4  O  X  X  X  O  O  O  O
 O  O  O  X  O  O  O  O  O  O
 O  O  O  O  O  B  B  O  O  O
 X  X  O  O  O  B  B  O  O  X
 O  O  O  O  O  O  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

```
Step: 2
 I  I  O  O  O  O  O  O  O  O
 I  I  O  O  O  O  O  O  O  O
 1  2  O  X  X  X  O  O  O  O
 3  4  O  X  O  O  O  O  O  O
 O  O  O  O  O  B  B  O  O  O
 X  X  O  O  O  B  B  O  O  X
 O  O  O  O  O  O  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

```
Step: 3
 I  I  O  O  O  O  O  O  O  O
 I  I  O  O  O  O  O  O  O  O
 O  O  O  X  X  X  O  O  O  O
 1  2  O  X  O  O  O  O  O  O
 3  4  O  O  O  B  B  O  O  O
 X  X  O  O  O  B  B  O  O  X
 O  O  O  O  O  O  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

```
Step: 4
 I  I  O  O  O  O  O  O  O  O
 I  I  O  O  O  O  O  O  O  O
 O  O  O  X  X  X  O  O  O  O
 3  O  O  X  O  O  O  O  O  O
 1  2  4  O  O  B  B  O  O  O
 X  X  O  O  O  B  B  O  O  X
 O  O  O  O  O  O  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

```
Step: 5
 I  I  O  O  O  O  O  O  O  O
 I  I  O  O  O  O  O  O  O  O
 O  O  O  X  X  X  O  O  O  O
 O  O  O  X  O  O  O  O  O  O
 3  1  2  O  O  B  B  O  O  O
 X  X  4  O  O  B  B  O  O  X
 O  O  O  O  O  O  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

```
Step: 6
 I  I  O  O  O  O  O  O  O  O
 I  I  O  O  O  O  O  O  O  O
 O  O  O  X  X  X  O  O  O  O
 O  O  O  X  O  O  O  O  O  O
 O  3  1  O  O  B  B  O  O  O
 X  X  2  O  O  B  B  O  O  X
 O  O  4  O  O  O  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

```
Step: 7
 I  I  O  O  O  O  O  O  O  O
 I  I  O  O  O  O  O  O  O  O
 O  O  O  X  X  X  O  O  O  O
 O  O  O  X  O  O  O  O  O  O
 O  O  3  1  O  B  B  O  O  O
 X  X  O  2  O  B  B  O  O  X
 O  O  O  4  O  O  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

```
Step: 8
 I  I  O  O  O  O  O  O  O  O
 I  I  O  O  O  O  O  O  O  O
 O  O  O  X  X  X  O  O  O  O
 O  O  O  X  O  O  O  O  O  O
 O  O  O  3  1  B  B  O  O  O
 X  X  O  O  2  B  B  O  O  X
 O  O  O  O  4  O  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

```
Step: 9
 I  I  O  O  O  O  O  O  O  O
 I  I  O  O  O  O  O  O  O  O
 O  O  O  X  X  X  O  O  O  O
 O  O  O  X  O  O  O  O  O  O
 O  O  O  O  3  1  B  O  O  O
 X  X  O  O  O  2  B  O  O  X
 O  O  O  O  O  4  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

```
Step: 10
 I  I  O  O  O  O  O  O  O  O
 I  I  O  O  O  O  O  O  O  O
 O  O  O  X  X  X  O  O  O  O
 O  O  O  X  O  O  O  O  O  O
 O  O  O  O  O  3  1  O  O  O
 X  X  O  O  O  4  2  O  O  X
 O  O  O  O  O  O  O  X  O  O
 O  O  O  O  O  O  X  X  O  O
 O  O  O  O  O  O  O  O  G  G
 O  O  O  O  X  O  O  O  G  G
```

Figure 10. The first ten steps from ‘I’ to ‘B’.

(Word Count: 3678)

Reference

Korf, R. E., 2010. *Artificial intelligence search algorithms* (pp. 22-22). Chapman & Hall/CRC.

Russell, S., & Norvig, P., 2005. AI a modern approach. *Learning*, *2*(3), 4.