Foundation of AI

Laiyuan Zhang
31035698
lz4y19@soton.ac.uk

December 2019

Contents

1 Approach

To visualize search methods, I used a two-dimensional array to represent the Blocksworld. I also created a class Node to represent the node of the tree and it includes the state of Blocksworld, the coordinate of agent, the depth of node and the heuristic cost which used for A* search. To achieve the position swap, I defined a direction array which contains 4 elements (1,0),(-1,0),(0,1),(0,-1), then we could use the coordinate of the agent to add these elements to realize the movement. After defining these basic types and methods, we could implement different search methods by using them.

For each method, we all need the initial state of Blocksworld as the root node of the tree. For breadth first search, since it traverses horizontally, we could use Queue to realize it cause it has the first-in, first-out characteristic. Each time we removed the head of the queue and judge if it is the target state. If not, we will expand the node and add the child nodes into the queue. Because of the characteristic, we could make sure it will iterate through all the nodes at the same depth.

Compared with breadth fist search, I choose the Deque to implement depth first search cause Deque allows us to remove the last node from the list just like Stack (For reasons of preference, I chose the Deque). Each time we removed the last of the Deque and check if it is the target state. If not, we will also expand the node and add the child nodes into the Deque, loop until we find the target state. To prevent meaningless loops, I will shuffle the child nodes before add them into the Deque.

The iterative deepening search is an improved algorithm of depth first search. Based on the depth first search, we limit the search depth. In the beginning, we will start from the depth of 0 which is the root node and if it is not the target state, the depth will increment to 1 and then continue the depth first search from the initial state. Through this iterative way to find the target.

Because A* heuristic search needs to calculate the cost to the target state, I used the PriorityQueue. By calculating the sum of cost so far and the estimated cost to target state to sort the nodes in the queue. Each time, we will remove the node from the PriorityQueue which has the lowest sum to check if it is the target state. If not, we will add its child nodes into PriorityQueue and resort, then remove the node has the lowest sum again. Loop until we find the target state.

2 Evidence

We use 6 to represent empty block, 1 to represent A, 2 to represent B, 3 to represent C, 0 to represent agent. Here is an example:

$$\begin{matrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 3 & 0 \end{matrix}$$

2.1 Breadth First Search Evidence

1. Add root to the queue: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 3 & 0 \end{bmatrix}$
2. Get the head from the queue: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 3 & 0 \end{bmatrix}$
3. Move the agent: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 0 \\ 1 & 2 & 3 & 6 \end{bmatrix}$ $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 0 & 3 \end{bmatrix}$
4. Get the head from the queue again: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 0 \\ 1 & 2 & 3 & 6 \end{bmatrix}$
5. Move the agent: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 3 & 0 \end{bmatrix}$ $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 0 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 3 & 6 \end{bmatrix}$ $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 0 & 6 \\ 1 & 2 & 3 & 6 \end{bmatrix}$
6. Get the head from the queue again: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 0 & 3 \end{bmatrix}$

7. Move the agent: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&0&6\\1&2&6&3\end{bmatrix}$ $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&2&3&0\end{bmatrix}$ $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&0&2&3\end{bmatrix}$

8.Loop until find the target state.(The position of agent does not matter) $\begin{bmatrix}6&6&6&6\\6&1&6&6\\0&2&6&6\\6&3&6&6\end{bmatrix}$

2.2 Deep First Search Evidence

1. Add root to the queue: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&2&3&0\end{bmatrix}$
2. Get the last from the Deque: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&2&3&0\end{bmatrix}$
3. Move the agent: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&0\\1&2&3&6\end{bmatrix}$ $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&2&0&3\end{bmatrix}$
4. Get the last from the queue again: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&0\\1&2&3&6\end{bmatrix}$
5. Move the agent: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&2&3&0\end{bmatrix}$ $\begin{bmatrix}6&6&6&6\\6&6&6&0\\6&6&6&6\\1&2&3&6\end{bmatrix}$ $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&0&6\\1&2&3&6\end{bmatrix}$
6. Get the last from the queue again: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&0&6\\1&2&3&6\end{bmatrix}$

7. Move the agent: $\begin{bmatrix}6&6&6&6\\6&6&0&6\\6&6&6&6\\1&2&3&6\end{bmatrix}$ $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&3&6\\1&2&0&6\end{bmatrix}$ $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&0&6&6\\1&2&3&6\end{bmatrix}$ $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&0\\1&2&3&6\end{bmatrix}$

8.Loop until find the target state.(The position of agent does not matter) $\begin{bmatrix}6&6&6&6\\6&1&6&6\\0&2&6&6\\6&3&6&6\end{bmatrix}$

2.3 Iterative Deepening Search Evidence

1. Add root to the queue: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&2&3&0\end{bmatrix}$

2. Get the last from the Deque: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&2&3&0\end{bmatrix}$

Can not expand because the max depth is 0. Now the Deque has been empty, then the max depth add 1 and put the root into the Deque again.

3. Get the last from the Deque: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&2&3&0\end{bmatrix}$

4. Move the agent: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&0\\1&2&3&6\end{bmatrix}$ $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&6\\1&2&0&3\end{bmatrix}$

5. Get the last from the queue again: $\begin{bmatrix}6&6&6&6\\6&6&6&6\\6&6&6&0\\1&2&3&6\end{bmatrix}$

Can not expand because the max depth is 1.

6. Get the last from the queue again: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 0 & 3 \end{bmatrix}$

Can not expand because the max depth is 1. Now the Deque has been empty, then the max depth add 1 again and put the root into the Deque again.

8.Loop until find the target state.(The position of agent does not matter) $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 1 & 6 & 6 \\ 0 & 2 & 6 & 6 \\ 6 & 3 & 6 & 6 \end{bmatrix}$

2.4 A* Search Evidence

1. Add root to the queue: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 3 & 0 \end{bmatrix}$

, total cost = cost so far + estimated cost to goal = 0 + 5 = 5

2. Get the last from the queue: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 3 & 0 \end{bmatrix}$

3. Move the agent: $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 0 \\ 1 & 2 & 3 & 6 \end{bmatrix}, totalcost = 1 + 5 = 6$ $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 1 & 2 & 0 & 3 \end{bmatrix}, totalcost = 1 + 6 = 7$

4. Get the lowest total cost from the queue again(if the left nodes share the same cost, get the first from the queue): $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 6 \\ 6 & 6 & 6 & 0 \\ 1 & 2 & 3 & 6 \end{bmatrix}$

5. Move the agent and calculate the total cost for each node. Loop until find the target state.(The position of agent does not matter) $\begin{bmatrix} 6 & 6 & 6 & 6 \\ 6 & 1 & 6 & 6 \\ 0 & 2 & 6 & 6 \\ 6 & 3 & 6 & 6 \end{bmatrix}$

3 Scalability

In the beginning, I tried to control the problem difficulty by changing the size of gird. However, when the size is 4*4, the BFS search has put quite a high pressure on my computer and it could not get the answer when the size is bigger than 5*5. Likely, DFS has the same problem when the size gets bigger. So, I chose to move the position of the agent to change the optimal solution depth to control problem difficulty. I recorded expanded node number and running time in the optimal solution depth from 8 - 14, did several tests for each method and use the average to draw line charts.

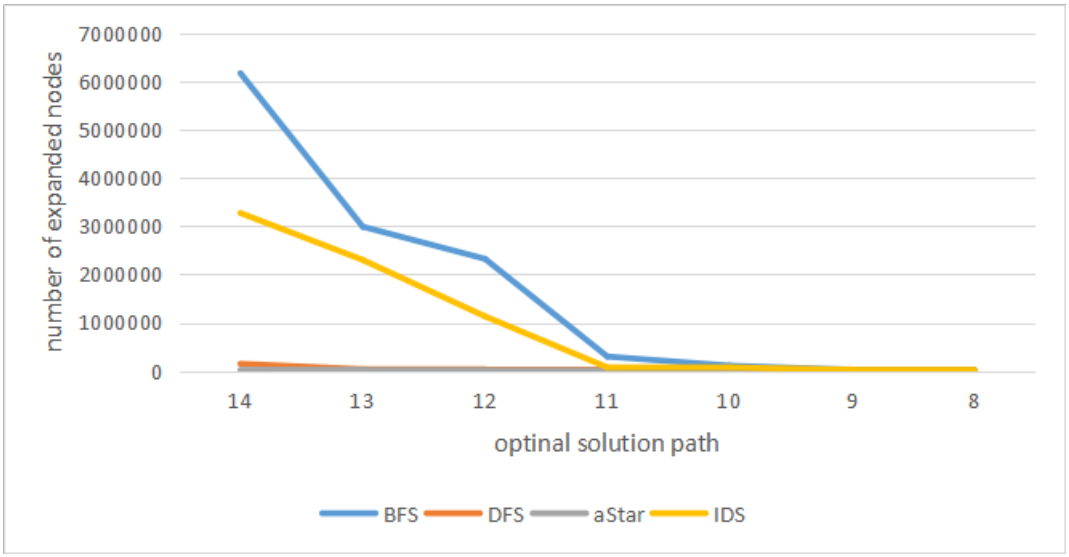

Figure 1:

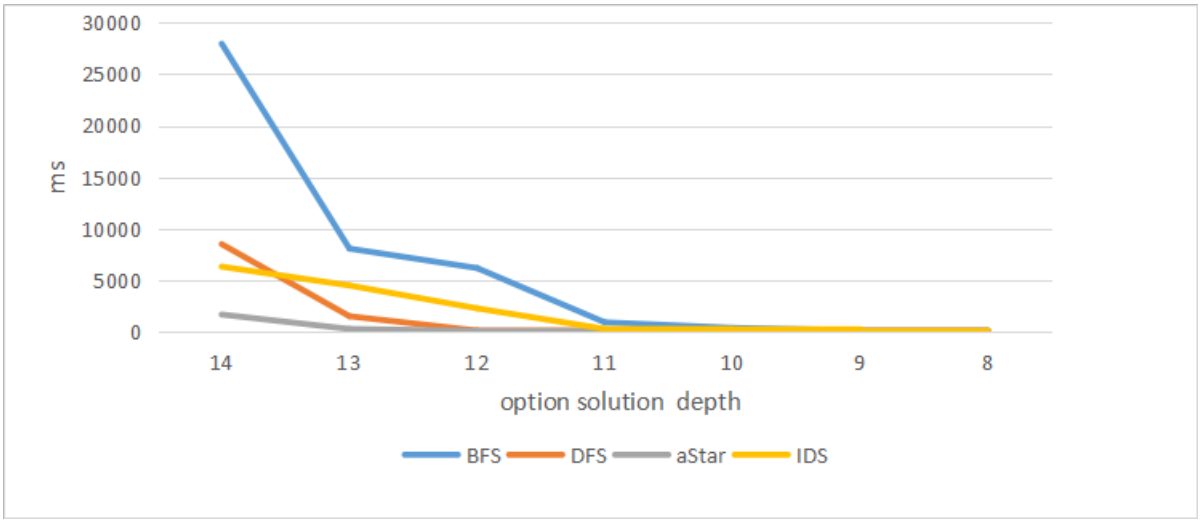

Figure 2:

As we can see from the figure 1, BFS always expanded the most number of nodes and aStar expanded the least number of nodes. This trend increases with the complexity of the problem.

From figure 2, generally, BFS still used the most time and aStar used the least time. The interesting thing is that, when the number of optional solution depth is bigger than 13, although the number of expanded nodes of IDS is more than DFS, the time is lower, which also improves IDS is an improved algorithm of DFS.

As a conclusion from these two figures, aStar is the best search methods among these four methods as it expands the least nodes and uses the least time and it could always get the best solution. On the other hand, BFS is the worst method as it uses the longest running time and put quite high pressure on the memory space. For DFS and IDS, although IDS will expand more nodes than DFS, however, when the problem gets more complicated, the number of expanded nodes influences a little. And IDS use the way of iteration to make it more effective. Although DFS is better than BFS, when it expanded the nodes, it is totally random, so sometimes it could not find the answer. I tried the size of 7*7 and it usually failed to find the solution.

4 Extras and limitations

4.1 Extras

During the experiment, when I use my own computer with 8G memory, it could not get the target because of the memory overflow problem. And when I studied it carefully, I found lots of memory used for the path that has been travelled which is meaningless. So, I tried to release these memory. I defined a string in Node class which is the string type of Blocksworld state, and I could easily use it to judge if it has been visited through the function *.contains*. In the search function, I define a HashSet to store the visited paths. Only the new child node will be added into the queue.

```
public class Node {
    int[][] board;
    String boardstring;
    int x_label;
    int y_label;
    int depth;

    //A*
    int heuristic;

    Node(int[][] B, int x, int y, int d) {
        board = B;
        boardstring = Arrays.deepToString(board);
        x_label = x;
        y_label = y;
        depth = d;

        heuristic = 0;
        int X = B.length;
        int Y = B[0].length;
```

Figure 3:

```
// keep the visited path
HashSet<String> haveDone = new HashSet();
haveDone.add(start.boardstring);
```

Figure 4:

```
// check if the node has been visited
if (!haveDone.contains(newNode.boardstring)) {
    nextLevelNodes.add(newNode);
    haveDone.add(newNode.boardstring);
}
```

Figure 5:

For BFS and DFS, these steps have improved the efficiency significantly. However, there is a little difference between IDS and these two search methods. Because we need to iterate, restart from the root for each depth, the HashSet which stores the visited path needs to be cleared when we increment the max depth to make sure it would not ignore some paths.

```
if (queue.size() == 0) {
    haveDone.clear();
    queue.add(start);
    haveDone.add(start.boardstring);

    maxDepth++;
}
```

Figure 6:

4.2 limitations

So far, if I want to change the size of Blocksworld, I need to write it by myself and it is fixed. In another word, the code is not reusable. For the convenience of future research, it will be better if it could generate the specified size of Blocksworld and random target state. This is one area I

can improve later. Another problem is that what I coded really depends on the CPU and memory space. I should improve the data structure which could spare the memory space. It might improve both space efficiency and time efficiency. The last is I could write a test class for testing, in this assignment, I just tested it for several times. It will be better if it could be tested automatically.

5 Code

5.1 Node.java

```
import java.util.Arrays;

public class Node {
    int[][] board;
    String boardstring;
    int x_label;
    int y_label;
    int depth;

    //A*
    int heuristic;

    Node(int[][] B, int x, int y, int d) {
       board = B;
        boardstring = Arrays.deepToString(board);
        x_label = x;
        y_label = y;
        depth = d;

        heuristic = 0;
        int X = B.length;
        int Y = B[0].length;
        for (int xx = 0; xx < X; ++xx)
              for (int yy = 0; yy < Y; ++yy) {
                  if (board[xx][yy] == 1)
                    heuristic += Math.abs(xx - 1) + Math.abs(yy -
```

```
                    1);
                if (board[xx][yy] == 2)
                  heuristic += Math.abs(xx - 2) + Math.abs(yy -
                    1);
                if (board[xx][yy] == 3)
                  heuristic += Math.abs(xx - 3) + Math.abs(yy -
                    1);
//                if (board[xx][yy] == 4)
//                  heuristic += Math.abs(xx - 4) + Math.abs(yy -
    1);
//                if (board[xx][yy] == 5)
//                  heuristic += Math.abs(xx - 5) + Math.abs(yy -
    1);
//                if (board[xx][yy] == 6)
//                    heuristic += Math.abs(xx - 6) + Math.abs(yy -
    1);
            }

    }
}
```

5.2 Main.java

```
public class Main extends Solution{

   public static void main(String[] args){
      int[][] board = {{6,6,6,6},{6,6,6,6},{6,6,6,6},{1,2,3,0}};
//    int[][] board =
    {{6,6,6,6,6},{6,6,6,6,6},{6,6,6,6,6},{6,6,6,6,6},{1,2,3,4,0}};
//    int[][] board =
    {{6,6,6,6,6,6},{6,6,6,6,6,6},{6,6,6,6,6,6},{6,6,6,6,6,6},{6,6,6,6,6,6},{1,2,3,4,5,0}
//    int[][] board =
    {{6,6,6,6,6,6,6},{6,6,6,6,6,6,6},{6,6,6,6,6,6,6},{6,6,6,6,6,6,6},{6,6,6,6,6,6,6},{6,
//    BFS(board);
      DFS(board);
//    aStar(board);
//    IDS(board);
```

```
    }
}
```

5.3 Solution.java

```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Deque;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Set;
import java.util.Stack;

public class Solution {

   public static int BFS(int[][] board) {
      long startTime = System.currentTimeMillis();
      long endTime;
      Node newNode = null;
      int nodesExpanded = 0;
      int X = board.length;
      int Y = board[0].length;
      // ax and ay is the position of agent
      int ax = 0;
      int ay = 0;
      // set directions
      int[][] directions = new int[][] { { 1, 0 }, { -1, 0 }, { 0,
          1 }, { 0, -1 } };
      Queue<Node> queue = new ArrayDeque();
      // look for the agent
      find: for (ax = 0; ax < X; ax++)
         for (ay = 0; ay < Y; ay++)
            if (board[ax][ay] == 0)
               break find;
```

```
Node start = new Node(board, ax, ay, 0);
queue.add(start);
// keep the visited path
HashSet<String> haveDone = new HashSet();
haveDone.add(start.boardstring);
while (!queue.isEmpty()) {
   ArrayList<Node> nextLevelNodes = new ArrayList<>();
   // get current position
   Node node = queue.remove();
   // for (int i = 0; i < 4; i++) {
   // for (int j = 0; j < 4; j++) {
   // System.out.print(node.board[i][j] + " ");
   // }
   // System.out.println();
   // }
   // System.out.println("depth is " + node.depth);
   // System.out.println("nodesExpanded is " + nodesExpanded);
   if (node.board[1][1] == 1 && node.board[2][1] == 2 &&
       node.board[3][1] == 3) {
      System.out.println("depth is " + node.depth);
      System.out.println("nodesExpanded is " + nodesExpanded);
      endTime = System.currentTimeMillis();
      System.out.println("running time is " + (endTime -
          startTime) + "ms");
      return node.depth;
   }
   nodesExpanded++;
   // move the agent
   for (int[] direction : directions) {
      int nei_x = direction[0] + node.x_label;
      int nei_y = direction[1] + node.y_label;

      if ((Math.abs(nei_x - node.x_label) + Math.abs(nei_y -
          node.y_label) != 1) || nei_x < 0 || nei_x >= X
            || nei_y < 0 || nei_y >= Y)
         continue;

      int[][] newboard = new int[X][Y];
      int t = 0;
```

```
            for (int[] row : node.board)
               newboard[t++] = row.clone();

            newboard[node.x_label][node.y_label] =
                newboard[nei_x][nei_y];
            newboard[nei_x][nei_y] = 0;

            newNode = new Node(newboard, nei_x, nei_y, node.depth +
                1);
            // check if the node has been visited
            // if (!haveDone.contains(newNode.boardstring)) {
            nextLevelNodes.add(newNode);
            // haveDone.add(newNode.boardstring);
            // }

         }
         Collections.shuffle(nextLevelNodes);
         for (Node nextLevelnode : nextLevelNodes) {
            if (nextLevelnode != null) {
               queue.add(nextLevelnode);
            }
         }

      }

      return -1;
   }

   public static int DFS(int[][] board) {
      long startTime = System.currentTimeMillis();
      int nodesExpanded = 0;
      int X = board.length;
      int Y = board[0].length;
      int ax = 0;
      int ay = 0;
      int[][] directions = new int[][] { { 1, 0 }, { -1, 0 }, { 0,
          1 }, { 0, -1 } };
      Deque<Node> queue = new ArrayDeque();
      find: for (ax = 0; ax < X; ax++)
         for (ay = 0; ay < Y; ay++)
```

```
      if (board[ax][ay] == 0)
         break find;

Node start = new Node(board, ax, ay, 0);
queue.add(start);

HashSet<String> haveDone = new HashSet();
haveDone.add(start.boardstring);
System.out.println(start.boardstring);

while (!queue.isEmpty()) {
   ArrayList<Node> nextLevelNodes = new ArrayList<>();
   Node node = queue.removeLast();
   for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4; j++) {
         System.out.print(node.board[i][j] + " ");
      }
      System.out.println();
   }
   System.out.println("depth is " + node.depth);
   System.out.println("nodesExpanded is " + nodesExpanded);

   if (node.board[1][1] == 1 && node.board[2][1] == 2 &&
       node.board[3][1] == 3) {
      System.out.println("depth is " + node.depth);
      System.out.println("nodesExpanded is " + nodesExpanded);
      long endTime = System.currentTimeMillis();
      System.out.println("running time is " + (endTime -
          startTime) + "ms");
      return node.depth;
   }
   nodesExpanded++;
   for (int[] direction : directions) {
      int nei_x = direction[0] + node.x_label;
      int nei_y = direction[1] + node.y_label;

      if ((Math.abs(nei_x - node.x_label) + Math.abs(nei_y -
          node.y_label) != 1) || nei_x < 0 || nei_x >= X
            || nei_y < 0 || nei_y >= Y)
         continue;
```

```
            int[][] newboard = new int[X][Y];
            int t = 0;

            for (int[] row : node.board)
               newboard[t++] = row.clone();

            newboard[node.x_label][node.y_label] =
                newboard[nei_x][nei_y];
            newboard[nei_x][nei_y] = 0;

            Node newNode = new Node(newboard, nei_x, nei_y,
                node.depth + 1);
            // if (!haveDone.contains(newNode.boardstring)) {
            nextLevelNodes.add(newNode);
            // haveDone.add(newNode.boardstring);
            // }
         }
         Collections.shuffle(nextLevelNodes);
         for (Node nextLevelnode : nextLevelNodes) {
            if (nextLevelnode != null) {
               queue.add(nextLevelnode);
            }
         }
      }

      return -1;
   }

   public static int aStar(int[][] board) {
      long startTime = System.currentTimeMillis();
      int nodesExpanded = 0;
      int X = board.length;
      int Y = board[0].length;
      int ax = 0;
      int ay = 0;
      int[][] directions = new int[][] { { 1, 0 }, { -1, 0 }, { 0,
          1 }, { 0, -1 } };
      PriorityQueue<Node> queue = new PriorityQueue<Node>(
            (a, b) -> (a.heuristic + a.depth) - (b.heuristic +
```

```
            b.depth));

find: for (ax = 0; ax < X; ax++)
   for (ay = 0; ay < Y; ay++)
      if (board[ax][ay] == 0)
         break find;
Node start = new Node(board, ax, ay, 0);
queue.add(start);

Set<String> haveDone = new HashSet();
haveDone.add(start.boardstring);

while (!queue.isEmpty()) {
   ArrayList<Node> nextLevelNodes = new ArrayList<>();
   Node node = queue.remove();
   System.out.println("string is " + node.boardstring);
   System.out.println("depth is " + node.depth);
   System.out.println("heuristic is " + node.heuristic);

   if (node.board[1][1] == 1 && node.board[2][1] == 2 &&
       node.board[3][1] == 3) {
      System.out.println("depth is " + node.depth);
      System.out.println("nodesExpanded is " + nodesExpanded);
      long endTime = System.currentTimeMillis();
      System.out.println("running time is " + (endTime -
          startTime) + "ms");
      return node.depth;
   }
   nodesExpanded++;
   for (int[] direction : directions) {
      int nei_x = direction[0] + node.x_label;
      int nei_y = direction[1] + node.y_label;

      if ((Math.abs(nei_x - node.x_label) + Math.abs(nei_y -
          node.y_label) != 1) || nei_x < 0 || nei_x >= X
            || nei_y < 0 || nei_y >= Y)
         continue;

      int[][] newboard = new int[X][Y];
      int t = 0;
```

```
            for (int[] row : node.board)
               newboard[t++] = row.clone();

            newboard[node.x_label][node.y_label] =
                newboard[nei_x][nei_y];
            newboard[nei_x][nei_y] = 0;

            Node newNode = new Node(newboard, nei_x, nei_y,
                node.depth + 1);
            // if (!haveDone.contains(newNode.boardstring)) {
            // haveDone.add(newNode.boardstring);
            nextLevelNodes.add(newNode);
            // }

         }
         for (Node nextLevelnode : nextLevelNodes) {
            if (nextLevelnode != null) {
               queue.add(nextLevelnode);
            }
         }
      }

      return -1;
   }

   public static int IDS(int[][] board) {
      long startTime = System.currentTimeMillis();
      int nodesExpanded = 0;
      int maxDepth = 0;
      int X = board.length;
      int Y = board[0].length;
      int ax = 0;
      int ay = 0;
      int[][] directions = new int[][] { { 1, 0 }, { -1, 0 }, { 0,
          1 }, { 0, -1 } };
      Deque<Node> queue = new ArrayDeque();
      Deque<Node> queueIter = new ArrayDeque();

      find: for (ax = 0; ax < X; ax++)
```

```
   for (ay = 0; ay < Y; ay++)
      if (board[ax][ay] == 0)
         break find;

Node start = new Node(board, ax, ay, 0);
queue.add(start);

Set<String> haveDone = new HashSet();
haveDone.add(start.boardstring);

while (!queue.isEmpty()) {
   ArrayList<Node> nextLevelNodes = new ArrayList<>();
   Node node = queue.removeLast();
   // System.out.println(node.boardstring);
   if (node.board[1][1] == 1 && node.board[2][1] == 2 &&
       node.board[3][1] == 3) {
      System.out.println("depth is " + node.depth);
      System.out.println("nodesExpanded is " + nodesExpanded);
      long endTime = System.currentTimeMillis();
      System.out.println("running time is " + (endTime -
          startTime) + "ms");
      return node.depth;
   } else if (node.depth < maxDepth) {
      nodesExpanded++;
      for (int[] direction : directions) {
         int nei_x = direction[0] + node.x_label;
         int nei_y = direction[1] + node.y_label;

         if ((Math.abs(nei_x - node.x_label) + Math.abs(nei_y
             - node.y_label) != 1) || nei_x < 0
               || nei_x >= X || nei_y < 0 || nei_y >= Y)
            continue;

         int[][] newboard = new int[X][Y];
         int t = 0;

         for (int[] row : node.board)
            newboard[t++] = row.clone();

         newboard[node.x_label][node.y_label] =
```

```
                    newboard[nei_x][nei_y];
                newboard[nei_x][nei_y] = 0;

                Node newNode = new Node(newboard, nei_x, nei_y,
                    node.depth + 1);
                // if (!haveDone.contains(newNode.boardstring)) {
                nextLevelNodes.add(newNode);
                // haveDone.add(newNode.boardstring);
                // }

             }

             for (Node nextLevelnode : nextLevelNodes) {
                if (nextLevelnode != null) {
                   queue.add(nextLevelnode);
                }
             }

          }
          if (queue.size() == 0) {
             haveDone.clear();
             queue.add(start);
             haveDone.add(start.boardstring);
             maxDepth++;
          }
       }

       return -1;
    }

}
```