School of Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

# Search Methods

## COMP2208 Intelligent Systems Assignment

Suyash Datt Dubey

Student ID: 29533414

November 29th, 2018

# Approach

For the assignment, the aim was to design methods to move an agent around a grid consisting of various blocks. The goal was to build a tower with 3 blocks named ‘A’, ‘B’ and ‘C’. The final position of the agent is irrelevant once the tower has been constructed. The agent is able to move up, down, left and right by swapping its position with the block in the respective direction. The Breadth-First Search, Depth-First Search, Iterative Deepening Search and A* Search algorithms are implemented to reach the goal state.

I programmed this assignment on Java as it has several data structures which proved to be extremely useful in the implementation of the algorithms.

The *State* is my implementation of the grid for the algorithms (except A*). It contains the size of the grid, the position of blocks and the current position of the agent int it. The grid itself has been implemented as an array containing characters (char[]). I have represented the agent as the ‘*’ symbol. The *moveAgent()* method handles the movement of the agent in the various directions. The *swap()* method performs the swapping of characters in the character array. The *goalStateReached()* method checks if the goal state has been reached. The *displayState()* method is used to print out the array in the form of a grid. The *getAgentPos()* method returns the position of the agent in the grid. Other methods are mostly getters.

The *StateAStar* is my implementation of the grid for A* algorithm. It is very similar to the *State* class. The only difference being that the grid has been implemented using a matrix of characters. Due to this change, some of the methods look a little different, but perform the exact same functions as those in the *State* class.

The *Node* is the object that is explored in the search algorithms (except A*). It contains a *State* object, a parent node, depth of the node, an integer representing the number of nodes expanded, and a String representing the direction of movement. A *getNeighbours()* method generates an ArrayList of possible neighbours of the node. A *getSolutionPath()* method returns the ArrayList of the moves made up to the current node. The *resetNodesExpanded()* sets the number of nodes expanded to 0. Other methods are mostly getter methods.

The *NodeAStar* is the object that is explored in the A* algorithm. It implements the Comparable class. It is very similar to the *Node* class. In addition to the other class variables, this class also contains an int *manhattanDistance*. The methods are also very similar and perform the same functions. One addition is the *createNewState()* method assists in adding new states in the *possibleMoves()* method. The other addition is the *calculateManhattanDistance()* method which generates the heuristic of the state in the node in relation to the goal state. The class also contains a version of the *compareTo()* which helps in comparing the heuristic of nodes. There are some getter methods present as well which assist in the calculation of the heuristic.

The *Main* class is used to test the various algorithms for multiple start states.

# Testing and Evidence of the 4 Search Methods

## Breadth-First Search (BFS)

The breadth-first search algorithm is implemented using the pseudocode from the sources in the references section. A method *bfs()* is called in the constructor of the class. The method first checks if the state in the root node passed to it is the goal state. If it isn't, a queue is created, and the root is added to the queue. While the queue is not empty, the first node is obtained from the queue. All the neighbours of this current node are generated. If the state of one of the neighbours of this node is the goal state, then it is returned. Otherwise, the neighbours are added to the queue. The loop continues until a node whose state matches the goal state is found. If no such state exists, then null is returned, and an error message is printed.

## Depth-First Search (DFS)

In this algorithm, a method *dfs()* is called in the constructor of the class. The method first creates a stack and adds the root node passed to it to the stack. While the stack is not empty, the first node is popped from the stack. If the state of the node is the same as that of the goal state, then it is returned. Otherwise, all the neighbours of this *current* node are generated and pushed on to the stack. The loop continues until a node whose state matches the goal state is found. If no such state exists, then null is returned, and an error message is printed.

## Iterative Deepening Search (IDS)

For this algorithm, depth-first search has been adapted to search only till a maximum depth, which keeps increasing iteratively. The root node is passed to *ids()* method when it is called in the constructor. A while loop runs where an integer *limit* is initially set to be 0. A node *result* represents the node returned by the *dls()* method. If the value of *result* is not null, the node *result* is returned. If the value of *result* is null then the value of *limit* is increased by 1 and the while loop continues until a not null node is found.

The *dls()* method is quite similar to the *dfs()* method in the depth-first search algorithm. First it checks if *limit* is 0. If it is, it checks if the state of the *current* node passed to it is the goal state. If it is, then the *current* node is returned. If the limit is greater than 0, then a list of all the neighbours of the *current* node is generated. A *Node* object *found* represents the value returned by the *dls()* method when it is called recursively for each of the neighbours. If the value of *found* is not null, it is returned. If the *current* node has no neighbours, then null is returned. If none of the 3 conditions are triggered, then null is returned as well.

## A* Search

A* algorithm's working is similar to that of breadth-first search, except for that fact that a priority queue has been used in A* Search. It uses heuristics (in this case the Manhattan Distance) to estimate the cost of moving from the state of the *current* node to the goal state. The heuristic is defined in the *NodeAStar* class. A version of the *compareTo()* function is also defined in the *NodeAStar* class so that a priority queue can be utilised for this algorithm.

The root node is passed to the *aStar()* method when it is called in the constructor of the class. The method first checks if the state in the root node passed to it is the goal state. If it isn't, a priority queue is created. Next, the heuristic for the root node is calculated using the *calculateManhattanDistance()* method defined in the node class. The root node is then added to the priority queue. While there are still nodes to be analysed in the priority queue, the first node is obtained from the priority queue. If the state of the current node is the same as the goal state, it is returned. Otherwise, the neighbours of this node are generated, their heuristics are calculated, and they are added to the priority queue. The loop continues until a node whose state matches the goal state is found. If no such state exists, then null is returned, and an error message is printed.

The number of nodes expanded, computational time (in seconds), memory used (in MB), the solution state, the solution path and the number of moves made to reach the goal state are also printed for each algorithm.

Running the *Main* class can be used to test the algorithms for the default problem, i.e., Scenario 1. Other scenarios can be tested by changing the first argument passed to BFS, DFS and IDS to *startState2* for Scenario 2 and *startState3* for Scenario 3 and that to A* to *startStateA2* for Scenario 2 and *startStateA3* for Scenario 3.

The goal state remains the same for all the following scenarios.

Goal State:

| | | | |
|---|---|---|---|
| | | | |
| | A | | |
| | B | | |
| | C | | ☺ |

The final position of the agent is irrelevant. (Agent is depicted as * in the output code).

## Scenario 1

Start State:

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| A | B | C | ☺ |

BFS

```
Nodes Expanded: 6179535
Time taken by BFS: 3.726 seconds
Memory used by BFS: 853.346168 MB

  A
* B
  C

Solution Path: [Up, Left, Left, Down, Left, Up, Right, Down, Right, Up, Up, Left, Down, Left]
No. of Moves made: 14
```

DFS

```
Nodes Expanded: 210970
Time taken by DFS: 0.032 seconds
Memory used by DFS: 36.977648 MB

  A
  B
  C *

Solution Path: [Up, Up, Up, Down, Left, Right, Up, Left, Right, Down, Down, Up, Left, Right, Down,
No. of Moves made: 66601
```

IDS

```
Nodes Expanded: 43440980
Time taken by IDS: 2.161 seconds
Memory used by IDS: 402.724136 MB

  A
* B
  C

Solution Path: [Up, Left, Left, Down, Left, Up, Right, Down, Right, Up, Up, Left, Down, Left]
No. of Moves made: 14
```

A*

```
Nodes Expanded: 56006
Time taken by A*: 0.042 seconds
Memory used by A*: 0.0 MB


  A
* B
  C

Solution Path: [Up, Left, Left, Down, Left, Up, Right, Down, Right, Up, Up, Left, Down, Left]
No. of Moves made: 14
```

## Scenario 2

Start State:

| | | | |
|---|---|---|---|
| | | | |
| | ☺ | A | |
| | B | | |
| | C | | |

BFS

```
Nodes Expanded: 4
Time taken by BFS: 0.001 seconds
Memory used by BFS: 0.0 MB

  A *
  B
  C

Solution Path: [Right]
No. of Moves made: 1
```

DFS

```
Nodes Expanded: 202702
Time taken by DFS: 0.096 seconds
Memory used by DFS: 32.867944 MB

* A
  B
  C

Solution Path: [Down, Left, Up, Right, Right, Down, Up, Down, Right, Left, Right, Left, Down, Left, Up,
No. of Moves made: 63893
```

IDS

```
Nodes Expanded: 15
Time taken by IDS: 0.0 seconds
Memory used by IDS: 0.0 MB

  A *
  B
  C

Solution Path: [Right]
No. of Moves made: 1
```

A*

```
Nodes Expanded: 4
Time taken by A*: 0.001 seconds
Memory used by A*: 0.0 MB


  A *
  B
  C

Solution Path: [Right]
No. of Moves made: 1
```

## Scenario 3

Start State:

| | | | |
|---|---|---|---|
| | | A | |
| | | | |
| | | C | B |
| | | | ☺ |

BFS

```
Nodes Expanded: 29768918
Time taken by BFS: 24.829 seconds
Memory used by BFS: 3856.847056 MB

  A *
  B
  C

Solution Path: [Left, Left, Up, Right, Right, Down, Left, Left, Up, Right, Up, Up, Left, Down, Right]
No. of Moves made: 15
```

DFS

```
Nodes Expanded: 374546
Time taken by DFS: 0.049 seconds
Memory used by DFS: 54.884712 MB
  *
  A
  B
  C

Solution Path: [Up, Left, Down, Up, Left, Up, Right, Down, Down, Up, Left, Up, Up, Down, Left, Up, Right, Left,
No. of Moves made: 118315
```

IDS

```
Nodes Expanded: 63872441
Time taken by IDS: 2.953 seconds
Memory used by IDS: 422.504136 MB

  A *
  B
  C

Solution Path: [Left, Left, Up, Right, Right, Down, Left, Left, Up, Right, Up, Up, Left, Down, Right]
No. of Moves made: 15
```

A*

```
Nodes Expanded: 210698
Time taken by A*: 0.095 seconds
Memory used by A*: 41.476648 MB


  A *
  B
  C

Solution Path: [Left, Left, Up, Right, Right, Down, Left, Left, Up, Right, Up, Up, Left, Down, Right]
No. of Moves made: 15
```

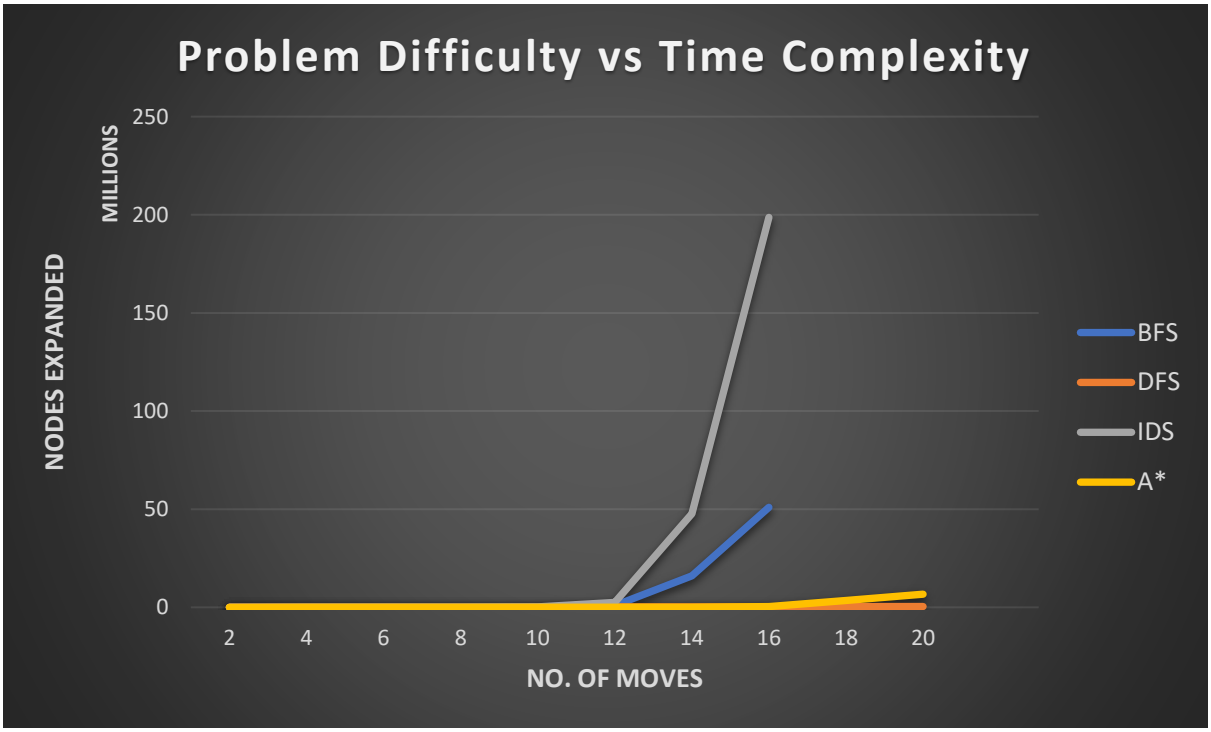
Conclusions and observations exploring various scenarios are described in the next section.

## Scalability Study

I performed the scalability study by changing up the positions of blocks in the start state so that the agent must make more moves to get to the goal state. The results are provided below:

| No. of Moves Made | Nodes Expanded | | | |
|---|---|---|---|---|
| | Breadth-First Search | Depth-First Search | Iterative Deepening Search | A* Search |
| 2 | 15 | 67817 | 30 | 18 |

| | | | | |
|---|---|---|---|---|
| 4 | 52 | 54591 | 431 | 61 |
| 6 | 905 | 116548 | 5406 | 394 |
| 8 | 17066 | 126883 | 70076 | 6351 |
| 10 | 61541 | 100501 | 253392 | 9927 |
| 12 | 887502 | 271341 | 2589178 | 53508 |
| 14 | 16078398 | 178526 | 47609591 | 103459 |
| 16 | 50946840 | 262124 | 198649033 | 408069 |
| 18 | Out of memory | 346412 | Out of memory | 3437753 |
| 20 | Out of memory | 422040 | Out of memory | 6707823 |




## Conclusions:

- BFS is the 2nd best for time/space complexity, even though it expands a large of nodes. It provides us with an optimal solution path. However, it starts to fail for more difficult problems. For my implementation, it started failing when more than 17 moves needed to be made to find the solution. This is because in BFS, every node at each depth is examined until the goal state is obtained. This is also why it takes a lot of time and memory. It is an effective algorithm to obtain an optimal solution for simple problems.
- DFS is probably the worst algorithm as it doesn't find an optimal solution and has poor time and space complexity. It is also very likely to go down a path which involves expansion of a greater number of nodes than required. It

expands lesser number of nodes as compared to BFS and is much quicker in my implementation as node selection has been randomised and hence the algorithm is more likely to make better choices while selecting neighbouring nodes. If nodes weren't selected randomly, DFS would take up a lot more time and space. It is suitable for small problems where reaching the goal state is more important then finding an optimal solution to it.

- IDS is better than DFS as it gives an optimal solution. It is still, however, inefficient as it expands a large number of nodes, even more than BFS. Therefore, its time and space complexities are much worse. It also begins to fail solving problems involving more than 17 moves in my implementation. It shouldn't be used for difficult problems.
- A* is the most efficient algorithm overall. It expands the fewest nodes, meaning it has the best time and space complexities among all the algorithms and provides us with an optimal solution. This is because it prioritises selecting nodes that are going roughly in the right direction by calculating the heuristic, the Manhattan Distance, before making a move. Hence it makes smarter moves which are closer to the goal state. It can also be improved by using better heuristic estimates. It is definitely the best algorithm to use for the Blocksworld Tile Puzzle.

# Extras and Limitations

## Extras

I have implemented an extra which is making the problem more difficult by introducing an unmoveable obstacle '#'. This can be tested by changing the first argument passed to each algorithm to *startStateExtra* for BFS, DFS and IDS and to *startStateAExtra* for A*. Some evidence has been pasted below.

Start State Tested:

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | # | |
| A | B | C | ☺ |

```
Breadth-First Search
Nodes Expanded: 2499515
Time taken by BFS: 0.574 seconds
Memory used by BFS: 365.745544 MB

  A
  B #
  C *

Solution Path: [Left, Left, Left, Up, Right, Down, Right, Right, Up, Up, Left, Left, Down, Down, Right]
No. of Moves made: 15

Depth-First Search
Nodes Expanded: 38158
Time taken by DFS: 0.014 seconds
Memory used by DFS: 7.950784 MB
  *
  A
  B #
  C

Solution Path: [Left, Left, Up, Up, Right, Up, Down, Up, Right, Down, Left, Right, Up, Left, Left, Left, Right, Down, Up, Down,
No. of Moves made: 13639
```

```
Iterative Deepening Search
Nodes Expanded: 10466394
Time taken by IDS: 0.679 seconds
Memory used by IDS: 400.089896 MB

  A
  B #
  C *

Solution Path: [Left, Left, Left, Up, Right, Down, Right, Right, Up, Up, Left, Left, Down, Down, Right]
No. of Moves made: 15

A* Search
Nodes Expanded: 28543
Time taken by A*: 0.024 seconds
Memory used by A*: 0.0 MB


  A
  B #
  C *

Solution Path: [Left, Left, Left, Up, Right, Down, Right, Right, Up, Up, Left, Left, Down, Down, Right]
No. of Moves made: 15
```

## Limitations

1. My implementation of the puzzle only works for problems of grid size 4x4 and cannot be extended to work for a small or larger grid.
2. BFS and IDS run out of space for more difficult problems in my implementation. I observed that they start to run out of heap space memory for problems where more than 17 moves need to be made.

# References


- Artificial Intelligence A Modern Approach 3rd Edition by Stuart J Russell and Peter Norvig
- Lecture Slides
- https://en.wikipedia.org/wiki/A*_search_algorithm
- https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search
- https://www.youtube.com/watch?v=7QcoJjSVT38

# Code

## Main class

```
public class Main {

    public static void main (String[] args){

        char[] startState = {' ', ' ', ' ', ' ',
                             ' ', ' ', ' ', ' ',
                             ' ', ' ', ' ', ' ',
                             'A', 'B', 'C', '*'};

        char[] startState2 = {' ', ' ', ' ', ' ',
                             ' ', '*', 'A', ' ',
                             ' ', 'B', ' ', ' ',
                             ' ', 'C', ' ', ' '};

        char[] startState3 = {' ', ' ', 'A', ' ',
                            ' ', ' ', ' ', ' ',
                            ' ', ' ', 'C', 'B',
                            ' ', ' ', ' ', '*'};

        char[] startStateExtra = {' ', ' ', ' ', ' ',
                                 ' ', ' ', ' ', ' ',
                                 ' ', ' ', '#', ' ',
                                 'A', 'B', 'C', '*'};

        char[] goalState = {' ', ' ', ' ', ' ',
                           ' ', 'A', ' ', ' ',
                           ' ', 'B', ' ', ' ',
                           ' ', 'C', ' ', '*'};

        char[][] startStateA = {{' ', ' ', ' ', ' '},
                                {' ', ' ', ' ', ' '},
                                {' ', ' ', ' ', ' '},
                                {'A', 'B', 'C', '*'}};

        char[][] startStateA2 = {{' ', ' ', ' ', ' '},
                                 {' ', '*', 'A', ' '},
                                 {' ', 'B', ' ', ' '},
                                 {' ', 'C', ' ', ' '}};

        char[][] startStateA3 = {{' ', ' ', 'A', ' '},
                                 {' ', ' ', ' ', ' '},
                                 {' ', ' ', 'C', 'B'},
                                 {' ', ' ', ' ', '*'}};

        char[][] startStateAExtra = {{' ', ' ', ' ', ' '},
                                     {' ', ' ', ' ', ' '},
                                     {' ', ' ', '#', ' '},
                                     {'A', 'B', 'C', '*'}};

        char[][] goalStateA = {{' ', ' ', ' ', ' '},
                               {' ', 'A', ' ', ' '},
```

```
                        {' ', 'B', ' ', ' '},
                        {' ', 'C', ' ', '*'}};

        System.out.println("Breadth-First Search");
        BreadthFirstSearch BFS = new BreadthFirstSearch(startState, goalState);
        Node.resetNodesExpanded();
        System.gc();
        System.out.println();

        System.out.println("Depth-First Search");
        DepthFirstSearch DFS = new DepthFirstSearch(startState, goalState);
        Node.resetNodesExpanded();
        System.gc();
        System.out.println();

        System.out.println("Iterative Deepening Search");
        IterativeDeepeningSearch IDS = new IterativeDeepeningSearch(startState, goalState);
        Node.resetNodesExpanded();
        System.gc();
        System.out.println();

        System.out.println("A* Search");
        AStarSearch AStar = new AStarSearch(startStateA, goalStateA);
        Node.resetNodesExpanded();
    }
}
```

## State class

```
public class State {

    private final Integer N;
    private char[] currentState;
    private final char agent = '*';
    private int agentPos;

    public State(char[] startState){
        this.N = startState.length;
        this.currentState = startState;
        agentPos = getAgentPos();
    }

    public State moveAgent (String direction){
        switch (direction) {
            case "up":
                swap(agentPos, agentPos - 4);
                agentPos -= 4;
                return this;
            case "down":
                swap(agentPos, agentPos + 4);
                agentPos += 4;
                return this;
            case "left":
                swap(agentPos, agentPos - 1);
```

```
                agentPos -= 1;
                return this;
            case "right":
                swap(agentPos, agentPos + 1);
                agentPos += 1;
                return this;
        }
        return null;
    }

    private char[] swap(int pos, int newPos){
        char temp = currentState[pos];
        currentState[pos] = currentState[newPos];
        currentState[newPos] = temp;
        return currentState;
    }

    public boolean goalStateReached(char[] goalState){
        for(int i = 0; i < N; i++){
            if (currentState[i] != 'A' && goalState[i] == 'A'){
                return false;
            }if (currentState[i] != 'B' && goalState[i] == 'B') {
                return false;
            }if(currentState[i] != 'C' && goalState[i] == 'C'){
                return false;
            }
        }
        return true;
    }

    public void displayState(){
        System.out.println(currentState[0] + " " + currentState[1] + " " + currentState[2] + " " +
currentState[3]);
        System.out.println(currentState[4] + " " + currentState[5] + " " + currentState[6] + " " +
currentState[7]);
        System.out.println(currentState[8] + " " + currentState[9] + " " + currentState[10] + " " +
currentState[11]);
        System.out.println(currentState[12] + " " + currentState[13] + " " + currentState[14] + " " +
currentState[15]);
    }

    public int getAgentPos(){
        for (int i = 0; i < N; i++) {
            if (currentState[i] == agent) {
                return i;
            }
        }
        return 0;
    }

    public char[] getCurrentState() {
        return currentState;
    }
}
```

# Node class

```
import java.util.*;

public class Node{

    private State state;
    private Node parent;
    private int depth = 0;
    private static int nodesExpanded = 0;
    private String direction;

    public Node(State startState){
        this.state = startState;
    }

    private Node(Node parent, State state, String direction){
        this.parent = parent;
        this.state = state;
        if(parent != null){
            this.depth = parent.depth + 1;
        }
        this.direction = direction;
        nodesExpanded++;
    }

    public ArrayList<Node> possibleMoves() {

        ArrayList<Node> neighbours = new ArrayList<>();
        int pos = state.getAgentPos();

        if (pos > 3 && (state.getCurrentState()[pos-4] != '#')) {
            State temp = new State(state.getCurrentState().clone());
            temp.moveAgent("up");
            neighbours.add(new Node(this, temp, "Up"));
        }if (pos < 12 && (state.getCurrentState()[pos+4] != '#')) {
            State temp = new State(state.getCurrentState().clone());
            temp.moveAgent("down");
            neighbours.add(new Node(this, temp, "Down"));
        } if (pos % 4 != 0 && (state.getCurrentState()[pos-1] != '#')) {
            State temp = new State(state.getCurrentState().clone());
            temp.moveAgent("left");
            neighbours.add(new Node(this, temp, "Left"));
        } if (pos % 4 != 3 && (state.getCurrentState()[pos+1] != '#')){
            State temp = new State(state.getCurrentState().clone());
            temp.moveAgent("right");
            neighbours.add(new Node(this, temp, "Right"));
        }
        return neighbours;
    }

    public ArrayList<String> getSolutionPath(){
        ArrayList<String> solutionPath = new ArrayList<>();
        Node node = this;
        while(node.parent != null){
            solutionPath.add(0, node.direction);
```

```
            node = node.parent;
        }
        return solutionPath;
    }

    public static void resetNodesExpanded(){
        nodesExpanded = 0;
    }

    public int getDepth() {
        return depth;
    }

    public int getNodesExpanded(){
        return nodesExpanded;
    }

    public State getState(){
        return state;
    }
}
```

## StateAStar class

```
public class StateAStar {

    private final Integer N;
    private char[][] currentState;
    private final char agent = '*';
    private Integer agentI;
    private Integer agentJ;

    public StateAStar(char[][] startState){
        this.N = startState.length;
        this.currentState = startState;
        agentI = getAgentI();
        agentJ = getAgentJ();
    }

    public StateAStar moveAgent (String direction){
        switch (direction) {
            case "up":
                swap(agentI, agentJ, agentI - 1, agentJ);
                agentI -= 1;
                return this;
            case "down":
                swap(agentI, agentJ, agentI + 1, agentJ);
                agentI += 1;
                return this;
            case "right":
                swap(agentI, agentJ, agentI, agentJ + 1);
                agentJ += 1;
                return this;
            case "left":
```

```
            swap(agentI, agentJ, agentI, agentJ - 1);
            agentJ -= 1;
            return this;
        }
        return null;
    }

    private char[][] swap(int i, int j, int temp1, int temp2){
        char temp = currentState[i][j];
        currentState[i][j] = currentState[temp1][temp2];
        currentState[temp1][temp2] = temp;
        return currentState;
    }

    public int getAgentI(){
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (currentState[i][j] == agent) {
                    return i;
                }
            }
        }
        return 0;
    }

    public int getAgentJ(){
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (currentState[i][j] == agent) {
                    return j;
                }
            }
        }
        return 0;
    }

    public boolean goalStateReached(char[][] goalState){
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (currentState[i][j] != 'A' && goalState[i][j] == 'A'){
                    return false;
                }if (currentState[i][j] != 'B' && goalState[i][j] == 'B') {
                    return false;
                }if(currentState[i][j] != 'C' && goalState[i][j] == 'C'){
                    return false;
                }
            }
        }
        return true;
    }

    public void displayState(){
        for (int i = 0; i < currentState.length; i++) {
            System.out.println();
            for (int j = 0; j < currentState[i].length; j++) {
                System.out.print(currentState[i][j] + " ");
```

```
            }
        }
        System.out.println();
    }

    public int getSize() {
        return N;
    }

    public char[][] getCurrentState() {
        return currentState;
    }
}
```

## NodeAStar class

```
import java.util.*;

public class NodeAStar implements Comparable{

    private StateAStar state;
    private NodeAStar parent;
    private int depth = 0;
    private String direction;
    private static int nodesExpanded = 0;
    private int manhattanDistance;


    public NodeAStar(StateAStar startState){
        this.state = startState;
    }

    private NodeAStar(NodeAStar parent, StateAStar state, String direction){
        this.parent = parent;
        this.state = state;
        if(parent != null){
            this.depth = parent.depth + 1;
        }
        this.direction = direction;
        nodesExpanded++;
    }

    public ArrayList<NodeAStar> possibleMoves() {
        ArrayList<NodeAStar> neighbours = new ArrayList<>();
        int i = state.getAgentI();
        int j = state.getAgentJ();

        if (i > 0 && state.getCurrentState()[i-1][j] != '#') {
            char [][] temp = createNewState(state.getCurrentState());
            StateAStar s = new StateAStar (temp);
            s.moveAgent("up");
            neighbours.add(new NodeAStar(this, s, "Up"));
        }if (i < state.getSize()-1 && state.getCurrentState()[i+1][j] != '#') {
            char[][] temp = createNewState(state.getCurrentState());
```

```
        StateAStar s = new StateAStar(temp);
        s.moveAgent("down");
        neighbours.add(new NodeAStar(this, s, "Down"));
    }if (j > 0 && state.getCurrentState()[i][j-1] != '#') {
        char [][] temp = createNewState(state.getCurrentState());
        StateAStar s = new StateAStar (temp);
        s.moveAgent("left");
        neighbours.add(new NodeAStar(this, s, "Left"));
    }if (j < state.getSize()-1 && state.getCurrentState()[i][j+1] != '#'){
        char [][] temp = createNewState(state.getCurrentState());
        StateAStar s = new StateAStar (temp);
        s.moveAgent("right");
        neighbours.add(new NodeAStar(this, s, "Right"));
    }
    return neighbours;
}

public ArrayList<String> getSolutionPath(){
    ArrayList<String> solutionPath = new ArrayList<>();
    NodeAStar node = this;
    while(node.parent != null){
        solutionPath.add(0, node.direction);
        node = node.parent;
    }

    return solutionPath;
}


private char[][] createNewState(char[][] startState){
    if (startState == null) return null;
    final char[][] newState = new char[startState.length][];

    for (int i = 0; i < startState.length; i++) {
        newState[i] = Arrays.copyOf(startState[i], startState[i].length);
    }
    return newState;
}

public void calculateManhattanDistance(char[][] goalState) {

    int aCostI = Math.abs(getA_I(goalState) - getA_I(state.getCurrentState()));
    int aCostJ = Math.abs(getA_J(goalState) - getA_J(state.getCurrentState()));
    int bCostI = Math.abs(getB_I(goalState) - getB_I(state.getCurrentState()));
    int bCostJ = Math.abs(getB_J(goalState) - getB_J(state.getCurrentState()));
    int cCostI = Math.abs(getC_I(goalState) - getC_I(state.getCurrentState()));
    int cCostJ = Math.abs(getC_J(goalState) - getC_J(state.getCurrentState()));

    this.manhattanDistance = aCostI + aCostJ + bCostI + bCostJ + cCostI + cCostJ + getDepth();
}

@Override
public int compareTo(Object node) {
    if(manhattanDistance < ((NodeAStar) node).manhattanDistance){
        return -1;
    }
```

```
    else if(manhattanDistance == ((NodeAStar) node).manhattanDistance){
        return 0;
    }
    else return 1;
}

private int getA_I(char[][] state){
    int N = state.length;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (state[i][j] == 'A') {
                return i;
            }
        }
    }
    return 0;
}

private int getA_J(char[][] state){
    int N = state.length;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (state[i][j] == 'A') {
                return j;
            }
        }
    }
    return 0;
}

private int getB_I(char[][] state){
    int N = state.length;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (state[i][j] == 'B') {
                return i;
            }
        }
    }
    return 0;
}

private int getB_J(char[][] state){
    int N = state.length;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (state[i][j] == 'B') {
                return j;
            }
        }
    }
    return 0;
}

private int getC_I(char[][] state){
    int N = state.length;
```

```
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (state[i][j] == 'C') {
                    return i;
                }
            }
        }
        return 0;
    }

    private int getC_J(char[][] state){
        int N = state.length;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (state[i][j] == 'C') {
                    return j;
                }
            }
        }
        return 0;
    }

    public int getDepth() {
        return depth;
    }

    public StateAStar getState(){
        return state;
    }

    public int getNodesExpanded(){
        return nodesExpanded;
    }
}
```

## BreadthFirstSearch class

```
import java.util.*;

public class BreadthFirstSearch {

    private char[] goalState;

    public BreadthFirstSearch(char[] startState, char[] goalState) {
        State state = new State(startState);
        this.goalState = goalState;
        bfs(new Node(state));
    }

    private Node bfs (Node node){

        long startTime = System.currentTimeMillis();
        long beforeUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
```

```
        if (node.getState().goalStateReached(goalState)) {
            System.out.println("Nodes Expanded: " + node.getNodesExpanded());

            double endTime = (System.currentTimeMillis() - startTime) / 1000.0;
            System.out.println("Time taken by BFS: " + endTime + " seconds");

            long afterUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
            double actualMemUsed = (afterUsedMem - beforeUsedMem)/1000000.0;
            System.out.println("Memory used by BFS: " + actualMemUsed + " MB");

            node.getState().displayState();
            return node;
        }

        Queue<Node> queue = new LinkedList<>();
        queue.add(node);

        while(!queue.isEmpty()){
            Node current = queue.poll();

            List<Node> neighbours = current.possibleMoves();

            for(Node n : neighbours){
                if (n.getState().goalStateReached(goalState)) {
                    System.out.println("Nodes Expanded: " + n.getNodesExpanded());

                    double endTime = (System.currentTimeMillis() - startTime) / 1000.0;
                    System.out.println("Time taken by BFS: " + endTime + " seconds");

                    long afterUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
                    double actualMemUsed = (afterUsedMem - beforeUsedMem)/1000000.0;
                    System.out.println("Memory used by BFS: " + actualMemUsed + " MB");

                    n.getState().displayState();

                    System.out.println();
                    System.out.println("Solution Path: " + n.getSolutionPath());
                    System.out.println("No. of Moves made: " + n.getSolutionPath().size());
                    return n;
                }
                queue.add(n);
            }
        }
        System.out.println("No solution was found.");
        return null;
    }
}
```

## DepthFirstSearch class

```
import java.util.*;

public class DepthFirstSearch {

    private char[] goalState;

    public DepthFirstSearch(char[] startState, char[] goalState) {
        State state = new State(startState);
        this.goalState = goalState;
        dfs(new Node(state));
    }

    private Node dfs(Node node){

        long startTime = System.currentTimeMillis();
        long beforeUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();

        Stack<Node> stack = new Stack<>();
        stack.add(node);

        while(!stack.isEmpty()){
            Node current = stack.pop();

            if (current.getState().goalStateReached(goalState)) {
                System.out.println("Nodes Expanded: " + current.getNodesExpanded());

                double endTime = (System.currentTimeMillis() - startTime) / 1000.0;
                System.out.println("Time taken by DFS: " + endTime + " seconds");

                long afterUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
                double actualMemUsed = (afterUsedMem - beforeUsedMem)/1000000.0;
                System.out.println("Memory used by DFS: " + actualMemUsed + " MB");

                current.getState().displayState();
                System.out.println();
                System.out.println("Solution Path: " + current.getSolutionPath());
                System.out.println("No. of Moves made: " + current.getSolutionPath().size());
                return current;
            }

            List<Node> neighbours = current.possibleMoves();
            Collections.shuffle(neighbours);

            for(Node n : neighbours){
                stack.push(n);
            }
        }
        System.out.println("No solution was found.");
        return null;
    }
}
```

# IterativeDeepeningSearch class

```
import java.util.*;

public class IterativeDeepeningSearch {

    private char[] goalState;

    public IterativeDeepeningSearch(char[] startState, char[] goalState) {
        State state = new State(startState);
        this.goalState = goalState;
        ids(new Node(state));
    }

    public Node dls(Node current, int limit, long startTime, long beforeUsedMem){

        if(limit == 0){
            if(current.getState().goalStateReached(goalState)) {
                System.out.println("Nodes Expanded: " + current.getNodesExpanded());

                double endTime = (System.currentTimeMillis() - startTime) / 1000.0;
                System.out.println("Time taken by IDS: " + endTime + " seconds");

                long afterUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
                double actualMemUsed = (afterUsedMem - beforeUsedMem)/1000000.0;
                System.out.println("Memory used by IDS: " + actualMemUsed + " MB");

                return current;
            }
        }if (limit > 0){
            List<Node> neighbours = current.possibleMoves();
            Collections.shuffle(neighbours);
            for (Node n: neighbours) {
                Node found = dls(n, limit-1, startTime, beforeUsedMem);
                if (found != null){
                    return found;
                }
            }
        }else if (current.possibleMoves() == null){
            return null;
        }
        return null;
    }

    public Node ids(Node node){

        long startTime = System.currentTimeMillis();
        long beforeUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();

        int limit = 0;

        while(true) {
            Node result = dls(node, limit, startTime, beforeUsedMem);
            limit++;
```

```
            if (result != null) {
                result.getState().displayState();
                System.out.println();
                System.out.println("Solution Path: " + result.getSolutionPath());
                System.out.println("No. of Moves made: " + result.getSolutionPath().size());
                return result;
            }
        }
    }
}
```

## AStarSearch class

```
import java.util.*;

public class AStarSearch {

    private char[][] goalState;

    public AStarSearch(char[][] startState, char[][] goalState) {
        StateAStar state = new StateAStar(startState);
        this.goalState = goalState;
        aStar(new NodeAStar(state));
    }

    private NodeAStar aStar(NodeAStar node){

        long startTime = System.currentTimeMillis();
        long beforeUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();

        if (node.getState().goalStateReached(goalState)) {
            System.out.println("Nodes Expanded: " + node.getNodesExpanded());

            double endTime = (System.currentTimeMillis() - startTime) / 1000.0;
            System.out.println("Time taken by A*: " + endTime + " seconds");

            long afterUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
            double actualMemUsed = (afterUsedMem - beforeUsedMem)/1000000.0;
            System.out.println("Memory used by A*: " + actualMemUsed + " MB");

            node.getState().displayState();
            return node;
        }

        PriorityQueue<NodeAStar> queue = new PriorityQueue<>();
        node.calculateManhattanDistance(goalState);
        queue.add(node);

        while(!queue.isEmpty()){
            NodeAStar current = queue.poll();

            if (current.getState().goalStateReached(goalState)) {
```

```
            System.out.println("Nodes Expanded: " + current.getNodesExpanded());

            double endTime = (System.currentTimeMillis() - startTime) / 1000.0;
            System.out.println("Time taken by A*: " + endTime + " seconds");

            long afterUsedMem = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
            double actualMemUsed = (afterUsedMem - beforeUsedMem)/1000000.0;
            System.out.println("Memory used by A*: " + actualMemUsed + " MB");

            current.getState().displayState();

            System.out.println();
            System.out.println("Solution Path: " + current.getSolutionPath());
            System.out.println("No. of Moves made: " + current.getSolutionPath().size()/2);
            return current;
          }

          List<NodeAStar> neighbours = current.possibleMoves();

          for(NodeAStar n : neighbours){
            n.calculateManhattanDistance(goalState);
            queue.add(n);
          }
        }
        System.out.println("No solution was found.");
        return null;
    }
}
```