AI Coursework Assignment – Blocksworld Tile Puzzle

COMP2208

Liyang Wang
29493145

Table of Contents

1. Approach

1.1 Setup

All four search methods share the same setup. First of all, each node carries 2 pieces of information: the current state of the board (agent & block positions) and action sequence it takes to get to that board state. In addition to that, there is also a controller class that handles moving the agent and updating the board state, as well as generating a list of possible moves.

1.2 Depth First Search (DFS)

For this problem, since there is no "bottom" of the tree, depth first search is implemented using an infinite while loop that only breaks and returns the action sequence when the board state of the node is the same as the goal state. Otherwise, every loop the controller generates a list of possible moves from the agent's current position and one move is randomly chosen.

1.3 Breath First Search (BFS)

At the start, the queue is initialized with the starting node which contains the starting board state and no action sequence. From there, at each loop, the function dequeues the node at the tail of the queue and checks if its board state is the same as the goal state. If so, then break the loop and return the action sequence. Otherwise, the controller generates all the possible moves and each move is used to generate a new successor and each successor is enqueued at the head of the queue.

1.4 Iterative Deepening Search (IDS)

At the start, just like BFS, the stack is initialized with the starting node that contains the starting board state and an empty action sequence. In the beginning of each loop, the node at the head of the stack is popped. If the node's board state matches the goal state, then the loop breaks and returns the result. Otherwise, if the length of the node's action sequence reaches the depth limit and the stack is empty, add one to the depth limit and reset the stack to the starting node. However, if the length of the action sequence reaches the depth limit but the stack is not empty, the search simply moves onto the next loop. Lastly, if the length of the action sequence is smaller than the depth limit, controller would generate the possible moves at that agent position and each move will generate a new successor which is then pushed onto the head of the stack.

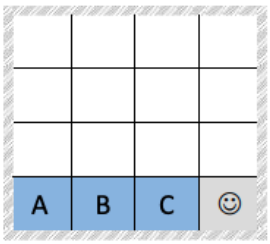
1.5 A* Search

A* is implemented almost exactly the same way as BFS, except for the use of a priority queue and heuristic function. In this case, the heuristic function I used is the sum of the Manhattan distance between the current board state to the goal state and the depth of the node. The reason for using the sum instead of just the Manhattan distance is for the heuristic function to represent the length of action sequence to get to the board state (shorter is better) and the distance to the goal. In the case of same priority, the node is picked based on the enqueue order.

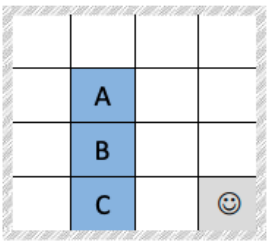
2. Evidence

2.1 The Problem

For each method, the problem used is the one given as an example from the assignment specification:



Start state.



Goal state.

2.2 Depth First Search (DFS)

Since the approach is randomized, the search is performed 3 times (appendix 1, 2, 3) in total to show that no matter how many nodes is expanded, the algorithm eventually finds a solution for this particular case. Due to the sheer size of the action sequence, it is impractical to draw out the actions. However, we can tell it did indeed find the action sequence to the goal state because the while loop only terminates and prints out the final action sequence if the goal state matches the current board state (appendix 4).

2.3 Breath First Search (BFS)

See appendix 5 for the illustration of the steps taken to get from the start state to the goal state. See appendix 6 for the debug output. At each depth expanded, a count of node in that depth is shown. When the solution is found, the final action sequence, the depth the action sequence resides. See appendix 7 for how the node is expanded.

2.4 Iterative Deepening Search (IDS)

Since action sequence IDS found is exactly the same as BFS, therefore the illustration can be found at appendix 5 as well. Debug output can be found in appendix 8, it is the exact same format as BFS. See appendix 9 for how the node expands.

2.5 A* Search

Similarly to IDS and BFS, the illustration of the action sequence can be found at appendix 5. Debug output that is the similar as IDS and BFS can be found at appendix 11 and node expansion can be found at appendix 12

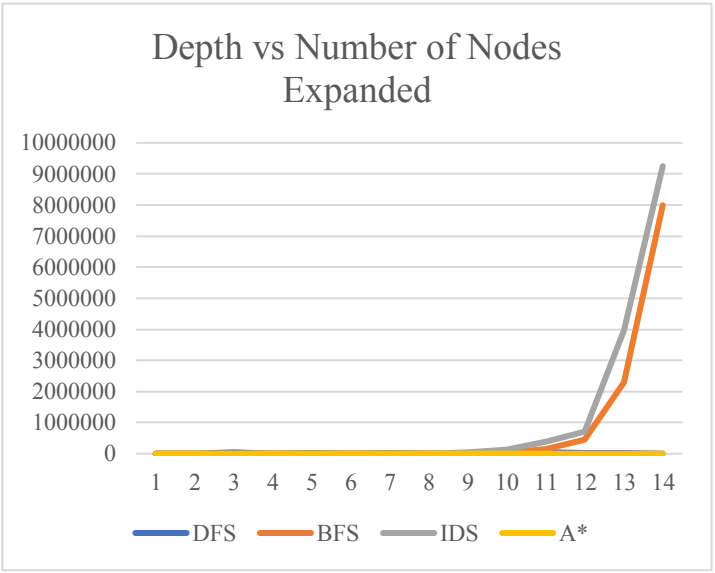
3. Scalability Study

3.1 Problem Difficulty Control

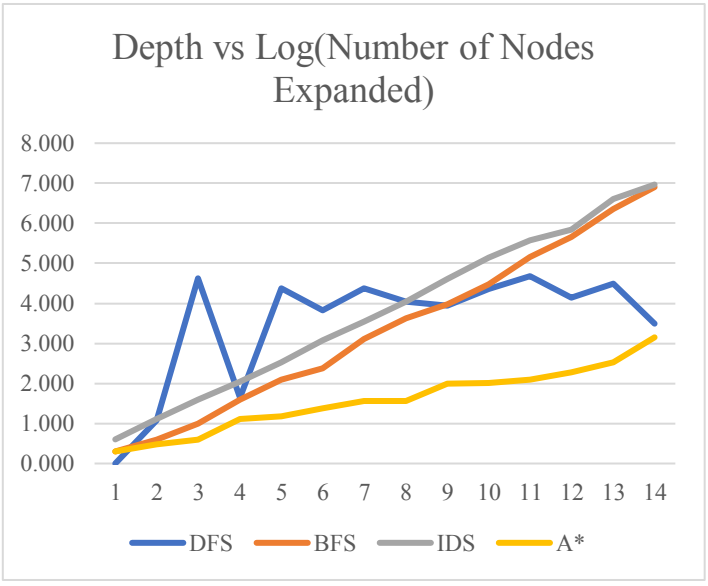
In order to do scalability test, since the time it takes for the search to finish is not a reliable source, the scalability test is done using the number of nodes expanded. I controlled the depth of the problem by keeping the starting state static and altering the goal state. Since each depth means a new action to the action sequence, limiting the depth can be done by coming up with new goal state that has the minimum number of steps to reaching it equating to the depth limit. See appendix 13 for goal state used.

3.2 The Test

Depth	DFS	BFS	IDS	A*
1	1	2	4	2
2	12	4	13	3
3	42727	10	40	4
4	44	40	109	13
5	23589	126	340	15
6	6818	236	1194	24
7	23263	1297	3479	36
8	11065	4192	11224	37
9	8831	9321	40153	98
10	22438	30273	137023	104
11	47707	141989	379629	122
12	13837	459483	700837	191
13	31417	2303459	3975165	340
14	3121	7992603	9247848	1415



Depth	DFS	BFS	IDS	A*
1	0.000	0.301	0.602	0.301
2	1.079	0.602	1.114	0.477
3	4.631	1.000	1.602	0.602
4	1.643	1.602	2.037	1.114
5	4.373	2.100	2.531	1.176
6	3.834	2.373	3.077	1.380
7	4.367	3.113	3.541	1.556
8	4.044	3.622	4.050	1.568
9	3.946	3.969	4.604	1.991
10	4.351	4.481	5.137	2.017
11	4.679	5.152	5.579	2.086
12	4.141	5.662	5.846	2.281
13	4.497	6.362	6.599	2.531
14	3.494	6.903	6.966	3.151



3.3 Conclusion

From the graph, one can clearly see that as the problem's difficulty (depth) scales up, the number of nodes expanded by A* scales up at a less dramatic rate as BFS and IDS, as both the BFS and IDS possess exponential time complexity. While in the worst-case scenario A* also has an exponential time complexity, because of the effectiveness of the heuristic function used (Manhattan distance + depth), many nodes are "pruned" away and the time complexity of A* is closer to logarithmic rather than exponential. Last but not least, since the DFS search is randomized, so is its scalability. From experiments, it can be seen that majority of the times the number of nodes expanded by DFS ranges from as little as one to as large as close to fifty thousand.

4. Extras and Limitations

4.1 Extras

In addition to what the problem asked for, I have also written an extension that allows for the algorithms to work when there is an obstacle in the grid. See section 5.2 for the code and appendix 14 for the action sequence output (the obstacle was set at (2, 1) and (2, 2)). Essentially, the main difference between working with obstacle and working without is how the successor nodes are generated. For without obstacle, the successor nodes (possible moves the agent can make at that given position) can simply be generated by checking whether the agent is touching one or more edges of the grid. However, when there are obstacles involved, what I did instead is to generate a list of all valid coordinates on the grid (all coordinates minus the obstacle ones). Then, successor node is generated by checking whether the movement would lead the agent off the board. Everything else stays the same.

4.2 Limitations

The biggest limitation of my approaches resides within the BFS. Unlike IDS where the nodes at the bottom of the tree does not need to generate any child (since the depth limit is hit, there is no point in more child nodes at the bottom), there is no "bottom" of the tree for BFS hence every node has to generate all of its child nodes which involves using the controller class to check for possible moves. This makes using BFS a lot more computationally expensive than IDS even though theoretically their time complexity should be the similar (exponential). Apart from that, the use of for loops and list comprehension throughout my code might have slowed down the searches even further. Lastly, in my extension, the controller function that gets all the possible moves has to check each move from the agent every time, which is slower than the standard approach where the possible moves is returned by checking whether the x or y coordinate of equates to zero or the grid edge length.

5. Code

5.1 Standard Version (No Obstacles)

Node class:

```python
import random
import time
import heapq


class Node:
    """
    A single Node carries information about the current state of the
    grid and the list of moves it takes to get there
    """
    def __init__(
            self,
            a_pos: tuple,
            b_pos: tuple,
            c_pos: tuple,
            agent_pos: tuple,
            moves_list: list,
    ):
        self.a_pos = a_pos
        self.b_pos = b_pos
        self.c_pos = c_pos
        self.agent_pos = agent_pos
        self.moves_list = moves_list
```

Controller Class:

```python
class Controller:
    def __init__(self, map_size: int):
        self.map_size = map_size

    # This takes in the current board positions and a move and returns the
    # new board positions after the move
    def move_agent(
            self,
            agent_pos: tuple,
            block_pos: list,
            move: str
    ) -> dict:
        a_pos = block_pos[0]
        b_pos = block_pos[1]
        c_pos = block_pos[2]

        prev_agent_pos = agent_pos
        new_agent_pos = None

        if move == 'left':
            new_agent_pos = (agent_pos[0] - 1, agent_pos[1])
        elif move == 'right':
            new_agent_pos = (agent_pos[0] + 1, agent_pos[1])
        elif move == 'up':
            new_agent_pos = (agent_pos[0], agent_pos[1] + 1)
        elif move == 'down':
            new_agent_pos = (agent_pos[0], agent_pos[1] - 1)

        if new_agent_pos == a_pos:
            a_pos = prev_agent_pos
        elif new_agent_pos == b_pos:
            b_pos = prev_agent_pos
        elif new_agent_pos == c_pos:
            c_pos = prev_agent_pos

        return {
            'block_pos': [a_pos, b_pos, c_pos],
            'agent_pos': new_agent_pos
        }
```

```
    # This takes in the agent's position and based on the map size returns
    # an array of all possible moves that can be made from that position
    def get_possible_moves(self, agent_pos: tuple) -> list:
        max_xy = self.map_size - 1
        agent_x = agent_pos[0]
        agent_y = agent_pos[1]

        if agent_x == 0 and agent_y == 0:
            return ['right', 'up']
        elif agent_x == max_xy and agent_y == max_xy:
            return ['left', 'down']
        elif agent_y == 0 and agent_x == max_xy:
            return ['left', 'up']
        elif agent_x == 0 and agent_y == max_xy:
            return ['right', 'down']
        elif agent_x == max_xy:
            return ['left', 'up', 'down']
        elif agent_y == max_xy:
            return ['left', 'right', 'down']
        elif agent_x == 0:
            return ['right', 'up', 'down']
        elif agent_y == 0:
            return ['up', 'left', 'right']
        else:
            return ['up', 'down', 'left', 'right']
```

DFS:

```
def dfs(map_size: int, starting_node: Node, goal_pos: list):
    controller = Controller(map_size)
    node = starting_node
    node_count = 0

    while True:
        potential_moves = controller.get_possible_moves(node.agent_pos)
        # Randomly choose a move and go
        move = random.choice(potential_moves)
        # New list of moves it takes to get to the new position
        moves_list = node.moves_list + [move]
        new_pos = controller.move_agent(node.agent_pos, [node.a_pos, node.b_pos, node.c_pos], move)
        node = Node(
            a_pos=new_pos['block_pos'][0],
            b_pos=new_pos['block_pos'][1],
            c_pos=new_pos['block_pos'][2],
            agent_pos=new_pos['agent_pos'],
            moves_list=moves_list
        )
        node_count += 1

        # If the new move leads to the goal position, return the list of moves
        if new_pos['block_pos'] == goal_pos:
            return {
                'moves': moves_list,
                'node_count': node_count
            }
```

BFS:

```python
def bfs(map_size: int, starting_node: Node, goal_pos: list):
    controller = Controller(map_size)
    node_count = 0
    depth = 0
    # Initialize the queue with the start position
    queue = [starting_node]

    while queue:
        node = queue.pop(0)
        node_count += 1

        if len(node.moves_list) - 1 > depth:
            depth += 1
            print(f'Depth: {depth}, Node count: {node_count}')

        if [node.a_pos, node.b_pos, node.c_pos] == goal_pos:
            return {
                'moves': node.moves_list,
                'node_count': node_count
            }
        else:
            possible_moves = controller.get_possible_moves(node.agent_pos)
            # Generate successors based on possible moves from the agent's position
            for move in possible_moves:
                new_pos = controller.move_agent(node.agent_pos, [node.a_pos, node.b_pos, node.c_pos], move)
                moves_list = node.moves_list + [move]

                queue.append(Node(
                    a_pos=new_pos['block_pos'][0],
                    b_pos=new_pos['block_pos'][1],
                    c_pos=new_pos['block_pos'][2],
                    agent_pos=new_pos['agent_pos'],
                    moves_list=moves_list
                ))
```

IDS:

```python
def ids(map_size: int, starting_node: Node, goal_pos: list):
    depth_limit = 0
    node_count = 0
    controller = Controller(map_size)
    stack = [starting_node]

    global depth_start_time
    depth_start_time = time.time()

    while stack:
        node = stack.pop()
        node_count += 1

        # Check if the current node is at the goal
        if [node.a_pos, node.b_pos, node.c_pos] == goal_pos:
            return {
                'moves': node.moves_list,
                'node_count': node_count
            }
        else:
            # Check if all the nodes at this depth limit has been checked
            if len(node.moves_list) == depth_limit and not stack:
                print(f'Depth: {depth_limit}, Node count: {node_count}')
                stack = [starting_node]
                depth_limit += 1
            # Only add new nodes to the queue if the current node isn't at the bottom of the tree
            elif len(node.moves_list) < depth_limit:
                possible_moves = controller.get_possible_moves(node.agent_pos)
                for move in possible_moves:
                    new_pos = controller.move_agent(node.agent_pos, [node.a_pos, node.b_pos, node.c_pos], move)
                    moves_list = node.moves_list + [move]

                    stack.append(Node(
                        a_pos=new_pos['block_pos'][0],
                        b_pos=new_pos['block_pos'][1],
                        c_pos=new_pos['block_pos'][2],
                        agent_pos=new_pos['agent_pos'],
                        moves_list=moves_list
                    ))
```

A*:

```python
def a_star(map_size: int, starting_node: Node, goal_pos: list):
    # Heuristic function
    def manhattan_distance(block_positions: list) -> int:
        output = 0

        for index, val in enumerate(block_positions):
            x_diff = abs(goal_pos[index][0] - val[0])
            y_diff = abs(goal_pos[index][1] - val[1])
            output += (x_diff + y_diff)

        return output

    controller = Controller(map_size)
    node_count = 0
    # This is used as a secondary comparator in case the manhattan distance is the same
    pq_sequence = 0
    depth = 0
    # Initialize the queue
    queue = [(
        manhattan_distance([starting_node.a_pos, starting_node.b_pos, starting_node.c_pos]),
        pq_sequence,
        starting_node
    )]
    # Turn the queue into a priority queue
    heapq.heapify(queue)

    while queue:
        node = heapq.heappop(queue)[2]
        node_count += 1

        if len(node.moves_list) - 1 > depth:
            depth += 1
            print(f'Depth: {depth}, Node count: {node_count}')

        # Check if the node is at goal state
        if [node.a_pos, node.b_pos, node.c_pos] == goal_pos:
            return {
                'moves': node.moves_list,
                'node_count': node_count
            }
        else:
            possible_moves = controller.get_possible_moves(node.agent_pos)
            for index, move in enumerate(possible_moves):
                new_pos = controller.move_agent(node.agent_pos, [node.a_pos, node.b_pos, node.c_pos], move)
                moves_list = node.moves_list + [move]
                pq_sequence += 1

                heuristic = manhattan_distance(new_pos['block_pos'])

                heapq.heappush(queue, (
                    # The addition of depth ensures the algo doesn't just run around in a circle
                    heuristic + depth,
                    pq_sequence,
                    Node(
                        a_pos=new_pos['block_pos'][0],
                        b_pos=new_pos['block_pos'][1],
                        c_pos=new_pos['block_pos'][2],
                        agent_pos=new_pos['agent_pos'],
                        moves_list=moves_list
                    )
                ))
```

5.2 Extended Code (Obstacle)

New Controller Class:

```
class Controller:
    def __init__(self, map_size: int, wall_coords: list):
        self.map_size = map_size
        all_coords = [(x, y) for x in range(self.map_size) for y in range(self.map_size)]
        self.valid_coords = set(all_coords) - set(wall_coords)

    # This takes in the current board positions and a move and returns the
    # new board positions after the move
    def move_agent(
            self,
            agent_pos: tuple,
            block_pos: list,
            move: str
    ) -> dict:
        a_pos = block_pos[0]
        b_pos = block_pos[1]
        c_pos = block_pos[2]

        prev_agent_pos = agent_pos
        new_agent_pos = None

        if move == 'left':
            new_agent_pos = (agent_pos[0] - 1, agent_pos[1])
        elif move == 'right':
            new_agent_pos = (agent_pos[0] + 1, agent_pos[1])
        elif move == 'up':
            new_agent_pos = (agent_pos[0], agent_pos[1] + 1)
        elif move == 'down':
            new_agent_pos = (agent_pos[0], agent_pos[1] - 1)

        if new_agent_pos == a_pos:
            a_pos = prev_agent_pos
        elif new_agent_pos == b_pos:
            b_pos = prev_agent_pos
        elif new_agent_pos == c_pos:
            c_pos = prev_agent_pos

        return {
            'block_pos': [a_pos, b_pos, c_pos],
            'agent_pos': new_agent_pos
        }

    # This takes in the agent's position and based on the map size returns
    # an array of all possible moves that can be made from that position
    def get_possible_moves(self, agent_pos: tuple, block_pos: list) -> list:
        all_moves = ['left', 'up', 'right', 'down']
        valid_moves = []

        for move in all_moves:
            new_agent_pos = self.move_agent(agent_pos, block_pos, move)['agent_pos']
            if new_agent_pos in self.valid_coords:
                valid_moves.append(move)

        return valid_moves
```

6. Appendix

6.1 Appendix 1 – DFS Attempt #1

```
Final action sequence: ['left', 'up', 'right', 'left', 'left', 'left', 'down', 'right', 'left', 'up', 'right', 'right', 'left', 'up', 'up', 'right', 'down', 'left', 'up', 'right', 'down', 'right', 'left',
Final depth reached: 3121
Number of nodes expanded: 3121
```

6.2 Appendix 2 – DFS Attempt #2

```
Final action sequence: ['up', 'left', 'right', 'up', 'down', 'down', 'up', 'up', 'up', 'down', 'left', 'left', 'left', 'up', 'right', 'left', 'right', 'down', 'left', 'up', 'right', 'down', 'down', 'left
Final depth reached: 105060
Number of nodes expanded: 105060
```

6.3 Appendix 3 – DFS Attempt #3

```
Final action sequence: ['left', 'right', 'left', 'left', 'left', 'right', 'up', 'right', 'down', 'up', 'down', 'right', 'left', 'up', 'up', 'down', 'down', 'left', 'left', 'up', 'down', 'right', 'up', 'd
Final depth reached: 4975
Number of nodes expanded: 4975
```

6.4 Appendix 4 – DFS Loop Break Statement

```python
# If the new move leads to the goal position, return the list of moves
if new_pos['block_pos'] == goal_pos:
    return {
        'moves': moves_list,
        'node_count': node_count
    }
```

6.5 Appendix 5 – From Start to Goal State

Start

Step 1

Step 2

Step 3

Step 4

Step 5

Step 6

Step 7

Step 8

Step 9

Step 10

Step 11

Step 12

Step 13

Step 14

6.6 Appendix 6 – BFS Output

6.7 Appendix 7 – BFS Node Expansion

Start

Depth 1

Child A

Child B

Depth 2

Child of A

Child of B

6.9 Appendix 9 – IDS Output

```
Final action sequence: ['up', 'left', 'left', 'down', 'left', 'up', 'right', 'down', 'right', 'up', 'up', 'left', 'down', 'left']
Final depth reached: 14
Number of nodes expanded: 9247848
```

6.10 Appendix 10 – IDS Node Expansion

Depth Limit 0

Reached depth limit, restart from top

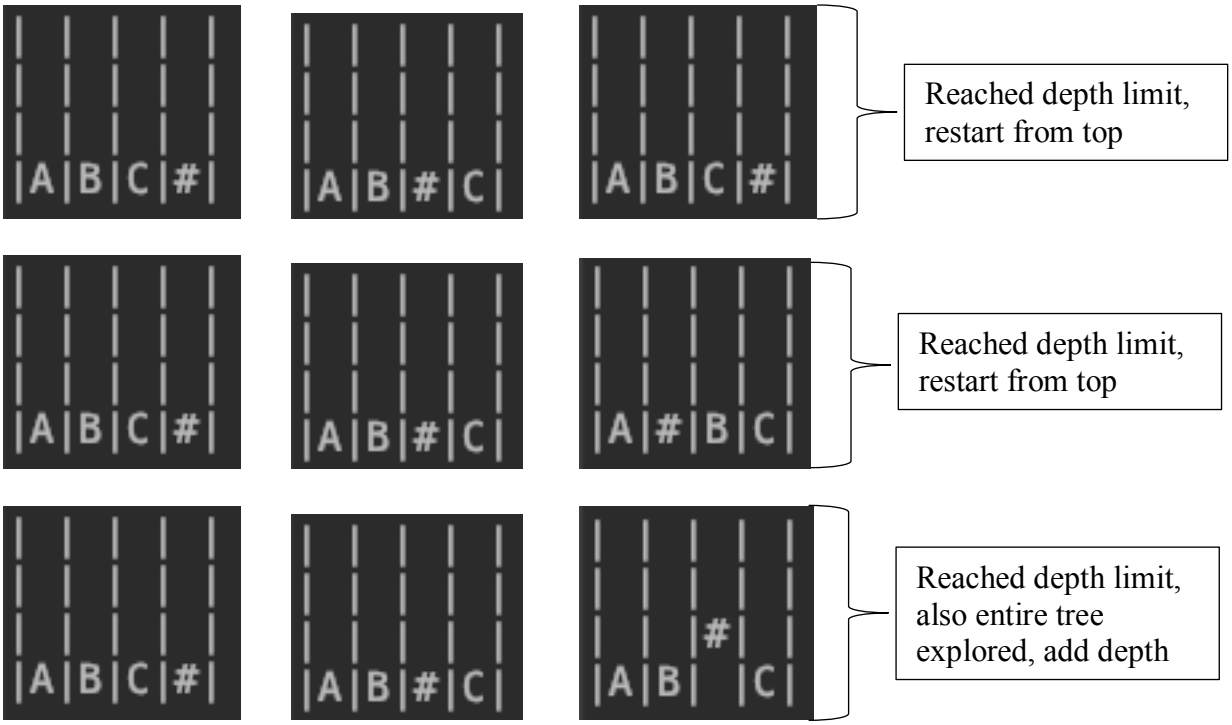
Reached depth limit, also entire tree explored, add depth

Depth Limit 1

Depth Limit 2

Reached depth limit, restart from top

Reached depth limit, restart from top


A B C #
A B # C
A B C #
Reached depth limit, restart from top
A B C #
A B # C
A # B C
Reached depth limit, restart from top
A B C #
A B # C
A B C
Reached depth limit, also entire tree explored, add depth

6.11 Appendix 11 – A* Output

```
Final action sequence: ['up', 'left', 'left', 'down', 'left', 'up', 'right', 'down', 'right', 'up', 'up', 'left', 'down', 'left']
Final depth reached: 14
Number of nodes expanded: 1415
```

6.12 Appendix 12 – A* Node Expansion

Start

PQ at Depth 1

Heuristic = 5 + 1

Heuristic = 6 + 1

PQ at Depth 2

Heuristic = 6 + 1

Heuristic = 5 + 2

Heuristic = 5 + 2

Heuristic = 5 + 2

Heuristic = 5 + 2

Heuristic = 6 + 2

Heuristic = 7 + 2

6.13 Appendix 13 – Goal States Used

1 Move

2 Moves

3 Moves

4 Moves

5 Moves

6 Moves

7 Moves

8 Moves

9 Moves

10 Moves

11 Moves

12 Moves

13 Moves

14 Moves

6.14 Appendix 14 – A* with Obstacle @ (2, 1), (2, 2) Output

```
Final action sequence: ['up', 'up', 'up', 'left', 'left', 'down', 'down', 'down', 'left', 'up', 'right', 'down', 'right', 'right', 'up', 'up', 'up', 'left', 'left', 'down', 'down', 'left']
Final depth reached: 22
Number of nodes expanded: 12070
```