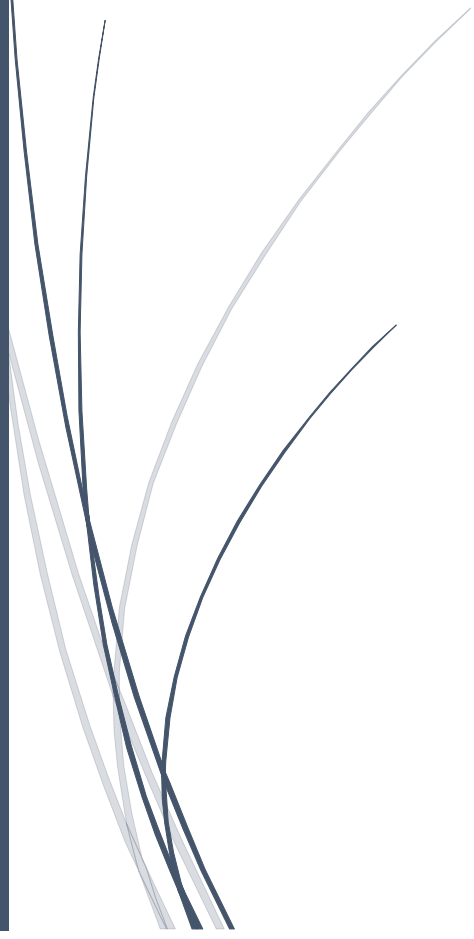
COS30019

3 Bottles Problem

Introduction to Artificial Intelligence



Noor Ul Ain Khurshid, 102763334

SEM 1, 2022

Contents

Problem Representation

The given problem is known as the "Three Bottles Problem." This problem includes three bottles which must reach a goal state or a final state from their initial states.

In the problem assigned here, there are three given bottles which can hold a limited amount of water. The first bottle (b1) can be filled with 10 litres of water, the second bottle (b2) can be filled with 6 litres of water, and the third bottle (b3) can be filled with 5 litres of water. The water supply is assumed to be limitless, and the bottles are ordinary bottles with no markings so it cannot be measured if the bottles are not filled to their maximum capacity.

State Representation

The state of the three bottles can be represented using a list [b1, b2, b3] = [x, y, z]. The water quantity in the bottles is denoted by the variables x, y, z respectively. Their ranges of are as follows:

$0 \leq x \leq 10$

$0 \leq y \leq 6$

$0 \leq z \leq 5$

Start State

The initial state of the three bottles from the assigned parameters is listed below:

a) Start: *b1 = 10, b2 = 0, b3 = 0*

b) Start: *b1 = 2, b2 = 0, b3 = 0*

c) Start: *b1 = 3, b2 = 0, b3 = 0*

d) Start: *b1 = 3, b2 = 0, b3 = 0*

*In state d), the maximum capacity of the bottles is changed to [b1: 11, b2: 7, b3: 4].

Goal State

The goal state denotes the final state of the bottles that is, how many litres of water should remain in the bottles at the end. Here, the final state of the bottles from the assigned parameters must be:

a) End: *b1 = 8, b2 = 0, b3 = 0*

b) End: *b1 = 4, b2 = 0, b3 = 0*

c) End: *b1 = 7, b2 = 0, b3 = 0*

d) End: *b1 = 7, b2 = 0, b3 = 0*

Operators

The goal state can be achieved by constructing a sequence of steps which the program will follow to get to the goal state from the initial state.

No.	Condition	Action	State
1	x < 10	fill b1:10 litre bottle	(10, y, z)
2	y < 6	fill b2:6 litre bottle	(x, 6, z)
3	z < 5	fill b3:5 litre bottle	(x y, 5)
4	x > 0	empty b1:10 litre bottle	(0, y, z)
5	y > 0	empty b2:6 litre bottle	(x, 0, z)
6	z > 0	empty b3:5 litre bottle	(x, y, 0)
7	(x + y) ≥ 6 & x > 0	transfer water from b1:10 litre to b2:6 litre, shift leftover to b1:10 Litre	(x - (6 - y), 6, z)
8	(x + z) ≥ 5 & x > 0	transfer water from b1:10 litre to b3:5 litre, shift leftover to b1:10 Litre	(x - (5 - z), y, 5)
9	(y + x) ≥ 10 & y > 0	transfer water from b2:6 litre to b1:10 litre, shift leftover to b2:6 litre	(10, y - (10 - x), z)
10	(y + z) ≥ 5 & y > 0	transfer water from b2:6 litre to b3:5 litre, shift leftover to b2:6 litre	(x, y - (5 - z), 5)
11	(z + x) ≥ 10 & z > 0	transfer water from b3:5 litre to b1:10 litre, shift leftover to b3:5 litre	(10, y, z - (10 - x))
12	(z + y) ≥ 6 & z > 0	transfer water from b3:5 litre to b2:6 Litre, shift leftover to b3:5 litre	(x, 6, z - (6 - y))
13	(x + y) ≤ 6 & x ≥ 0	transfer water from b1:10 litre to b2:6 litre, none remaining	(0, (x + y), z)
14	(x + z) ≤ 5 & x ≥ 0	transfer water from b1:10 litre to b3:5 litre, none remaining	(0, y, (x + z))
15	(y + x) ≤ 10 & y ≥ 0	transfer water from b2:6 litre to b1:10 litre, none remaining	((x + y), 0, z)
16	(y + z) ≤ 5 & y ≥ 0	transfer water from b2:6 litre to b3:5 litre, none remaining	(x, 0, (y + z))
17	(z + x) ≤ 10 & z ≥ 0	transfer water from b3:5 litre to b1:10 litre, none remaining	((x + z), y, 0)
18	(z + y) ≤ 6 & z ≥ 0	transfer water from b3:5 litre to b2:6 litre, none remaining	(x, (y + z), 0)

Search Strategies

Using artificial intelligence, there are a number of search strategies that can be implemented to solve the Three Bottle Problem. These search strategies can be categorized as informed and uninformed searches.

Informed Search

In informed search strategy, information can be attained such as the way to reach the final node, distance from final node, and path code. Informed search prioritizes efficiency by focusing on finding the final state instead spending time in search space.

The A* Search algorithm will be implemented to solve the Three Bottle Problem. Informed search strategy is also known as heuristic search.

Formula: $T(b) = O(b^d)$

Where, b = branching factor, $T(b)$ = Time complexity, $O(b^d)$ = Space complexity.

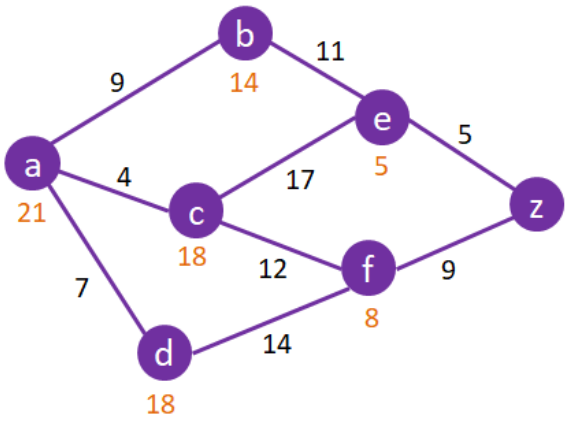


Figure 1: A search algorithm*

The A* search algorithm's efficiency depends on the given heuristics, and it provides a complete result in which may not be the shortest path, but it is definitely optimal in the case that the branching is limited.

Uninformed Search

The uninformed search is also known as the blind search algorithm. This strategy used the method of forcing through each node and it traverses the tree without providing any information, unlike informed search strategy.

The uninformed search strategies chosen are the depth first search and breadth first search strategies.

Depth First Search

In the depth first search, each brand node is traversed before moving to the next branch node. Depth first search implements stack to save the progress and the search continues until the tree reaches an end and the stack is null.

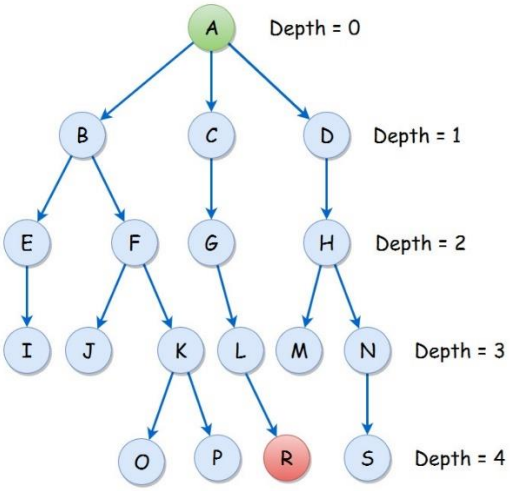


Figure 2: Depth first search

Formula: $T(b) = 1 + b^2 + b^3 + \ldots\ldots\ldots + b^m = O(b^m)$

$T(b)$ = time complexity, $O(b^m)$ = space complexity, m = maximum depth, b = node

Breadth First Search

Breadth first search uses the queue data structure by starting with an initial state node and inserts the visited node in the queue. The search goes on till the queue is empty and no nodes are left unvisited. Every next node to the selected node is visited and those nodes are inserted in the queue as well.

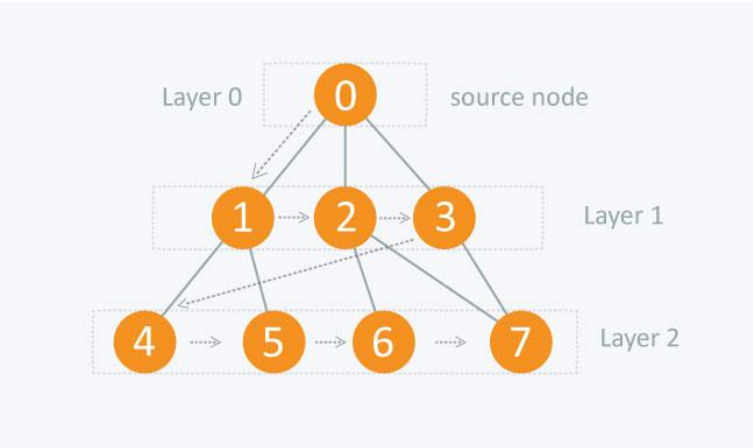


Figure 3: Breadth first search

Formula:

O (b^d) b + b2 + b3 + · · · + bd = O(b^d)

Where, b = node, d = shallowest depth, O(b^d) = space complexity

Search Tree

A* algorithm search

After an A* search strategy was implemented, the following tree was constructed with the initial state of [10, 0, 0] and final state of [8, 0, 0].

Figure 4: A tree diagram*

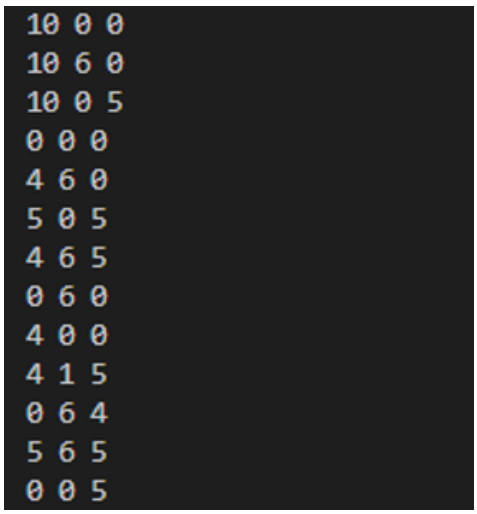


Figure 5: sequence

As aforementioned, the start state is [10, 0, 0]. In the priority queue, which is a queue where elements are given a certain priority and stored. The queue consists of the nodes to be visited and the information regarding the distance to the final node which is the path cost. It goes through the elements based on their priority. The elements with the smallest path cost are given priority and the new nodes are stored in the priority queue until all nodes are visited and the queue is empty.

Depth First Search

In implementing the depth first search, the following tree was constructed with the initial state of [10, 0, 0] and the final state of [8, 0, 0].

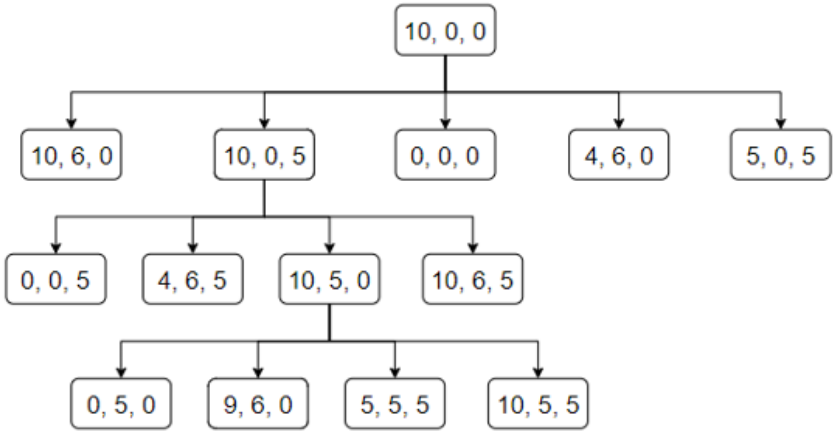


Figure 6: Depth first search tree

```
10 0 0
0 0 0
4 6 0
5 0 5
10 6 0
10 0 5
0 0 5
4 6 5
10 6 5
10 5 0
0 5 0
9 6 0
5 5 5
```

Figure 7: Sequence

Depth first search starts searching from the first node or the root node. The following states are added to the stack. One of the main differences between depth first search and breadth first search is that depth first search uses Last In First Out data structure whereas breadth first data structure used First In First Out data structure. The Last In First Out data structure goes through the queue starting from the last node appended to it and then visits the adjacent nodes which are not visited. The adjacent nodes are added to the stack which are explored, and this continues until all nodes are visited and no nodes are left in the stack. Since depth first search uses a stack data structure, it keeps going through the tree until the stack of nodes to be visited is empty and no nodes are left. It goes through the stack before entering a new branch of nodes.

Breadth First Search

In implementing the breadth first search, the following tree was constructed with the initial state of [10, 0, 0] and the final state of [8, 0, 0]. Breadth first search starts searching from the first node or the root node. This node is inserted into the queue and marked as visited. The search moves on to the next element which is not marked completed. The element goes through operators which results in more nodes which are appended in the queue. Unlike depth first search, the breadth first search uses the First In First Out queue. This data structures goes through all the nodes in the level because those nodes were appended first to the queue, before proceeding to the following level. This search continues until all nodes are visited or the final state is achieved.

Figure 8: Breadth first search

```
10 0 0
10 6 0
10 0 5
0 0 0
4 6 0
5 0 5
10 6 5
0 6 0
5 6 5
10 1 5
0 0 5
4 6 5
10 5 0
```

Figure 9: Sequence

Observations and Discussions

All three strategies, the A* search algorithm, the Breadth first search, the Depth first search were implemented on the given parameters and their results are shown below:

a) Start: *b1 = 10, b2 = 0, b3 = 0*

End: *b1 = 8, b2 = 0, b3 = 0*

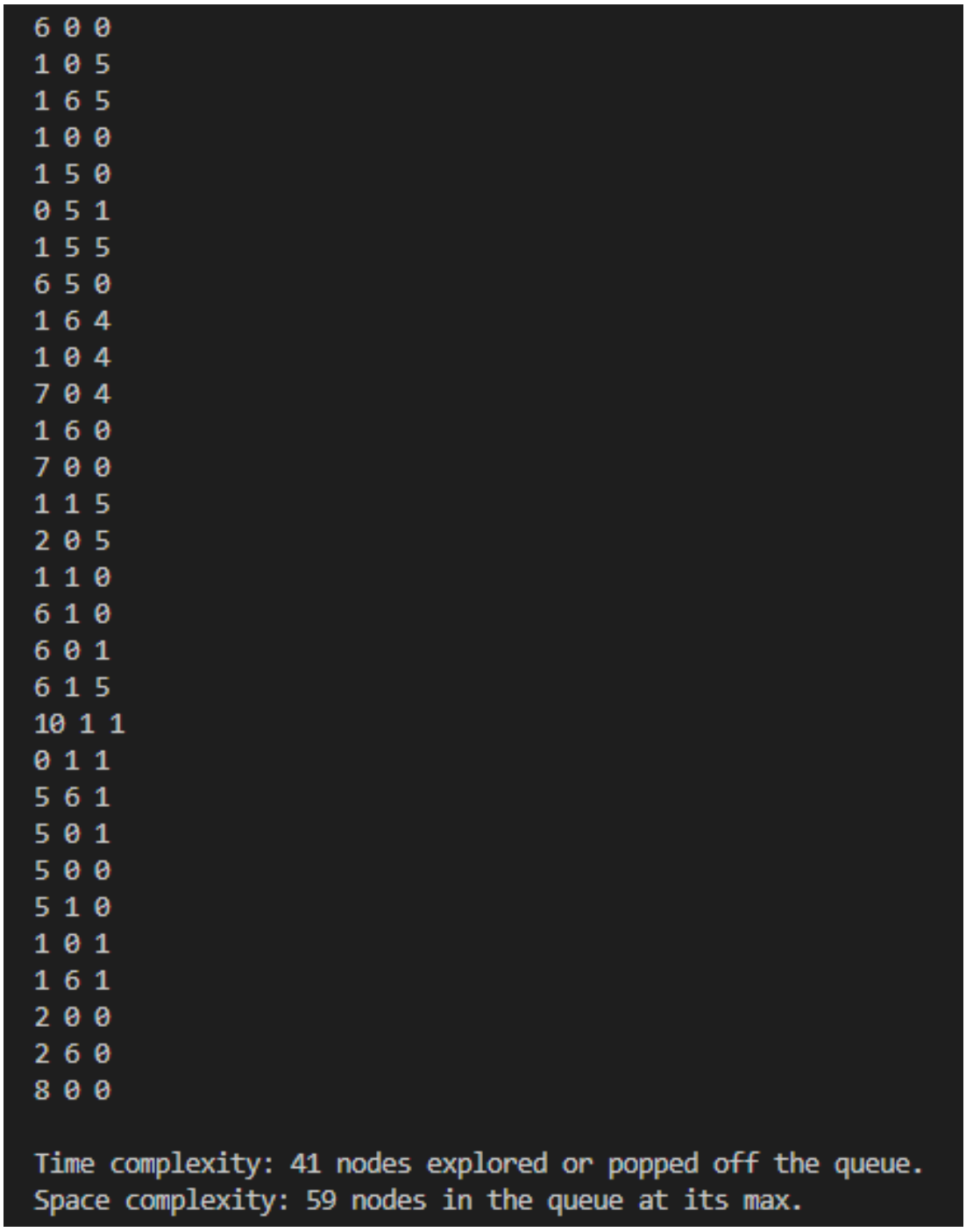


Figure 10: Depth first search

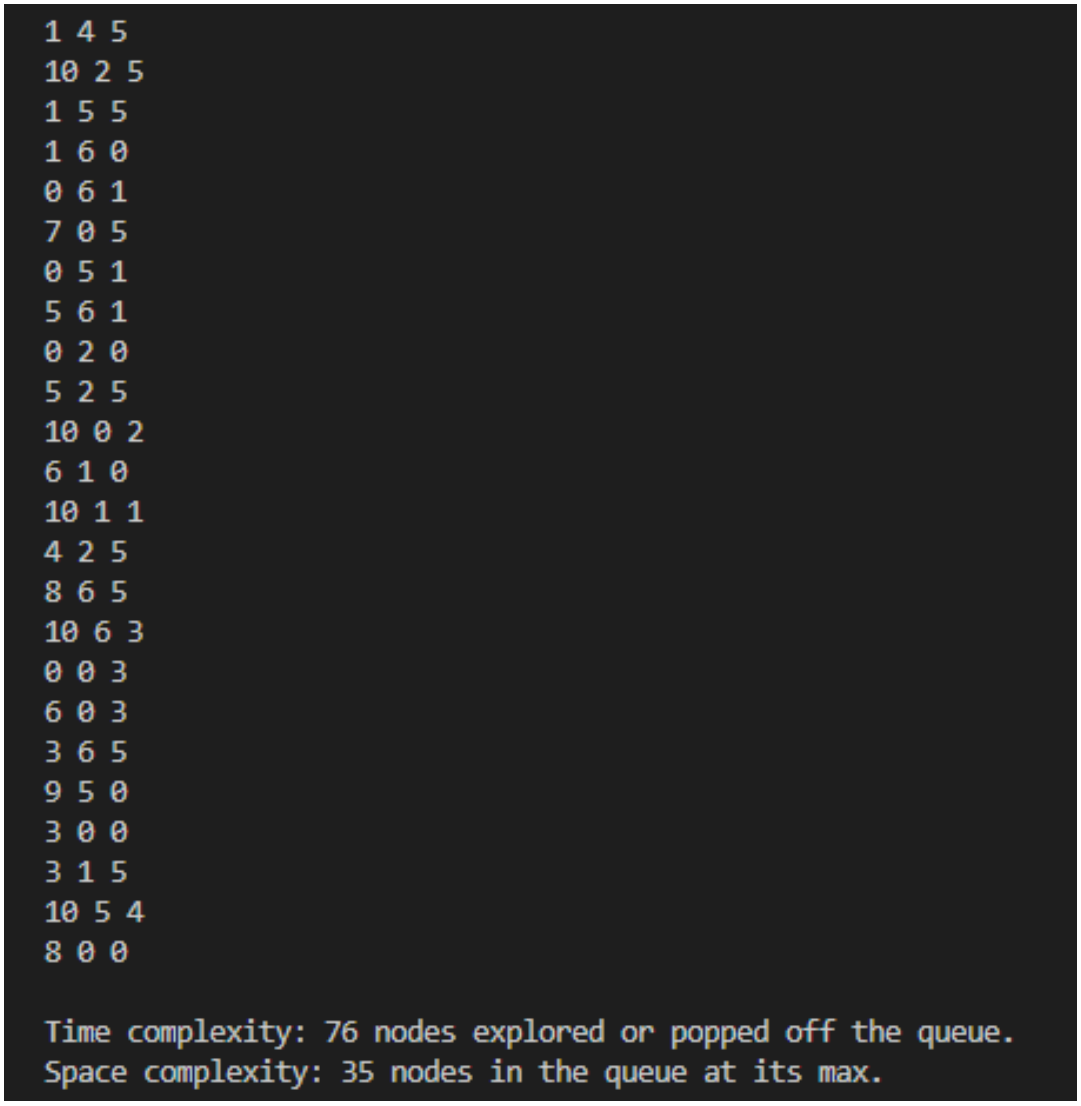


Figure 11: Breadth first search

```
Enter your choice: 3
10 0 0
10 6 0
10 0 5
0 0 0
4 6 0
5 0 5
10 6 5
0 0 5
4 6 5
10 5 0
10 5 5
0 5 0
9 6 0
5 5 5
9 6 5
0 6 0
9 0 0
9 1 5
9 0 5
3 6 0
4 0 5
3 6 5
10 0 4
9 5 0
10 6 4
0 0 4
4 6 4
10 4 0
9 5 5
8 6 0
4 5 5
10 4 5
0 4 0
5 4 5
8 6 5
8 0 0

Time complexity: 10 nodes explored or popped off the queue.
Space complexity: 24 nodes in the queue at its max.
```

Figure 12: A search algorithm*

b) Start: *b1 = 2, b2 = 0, b3 = 0*

End: *b1 = 4, b2 = 0, b3 = 0*

```
10 0 5
4 6 5
10 5 0
9 6 0
5 5 5
10 5 5
9 6 5
10 6 4
10 0 4
0 0 4
4 6 4
9 0 5
10 4 0
0 4 0
8 6 0
5 4 5
10 4 5
0 4 5
8 6 5
10 6 3
0 6 3
10 0 3
0 0 3
4 6 3
8 0 5
10 3 0
0 3 0
7 6 0
5 3 5
10 3 5
0 3 5
7 6 5
10 6 2
0 6 2
10 0 2
4 6 2
4 0 2
4 3 5
4 6 0
4 0 0

Time complexity: 25 nodes explored or popped off the queue.
Space complexity: 54 nodes in the queue at its max.
```

Figure 13: Depth first search

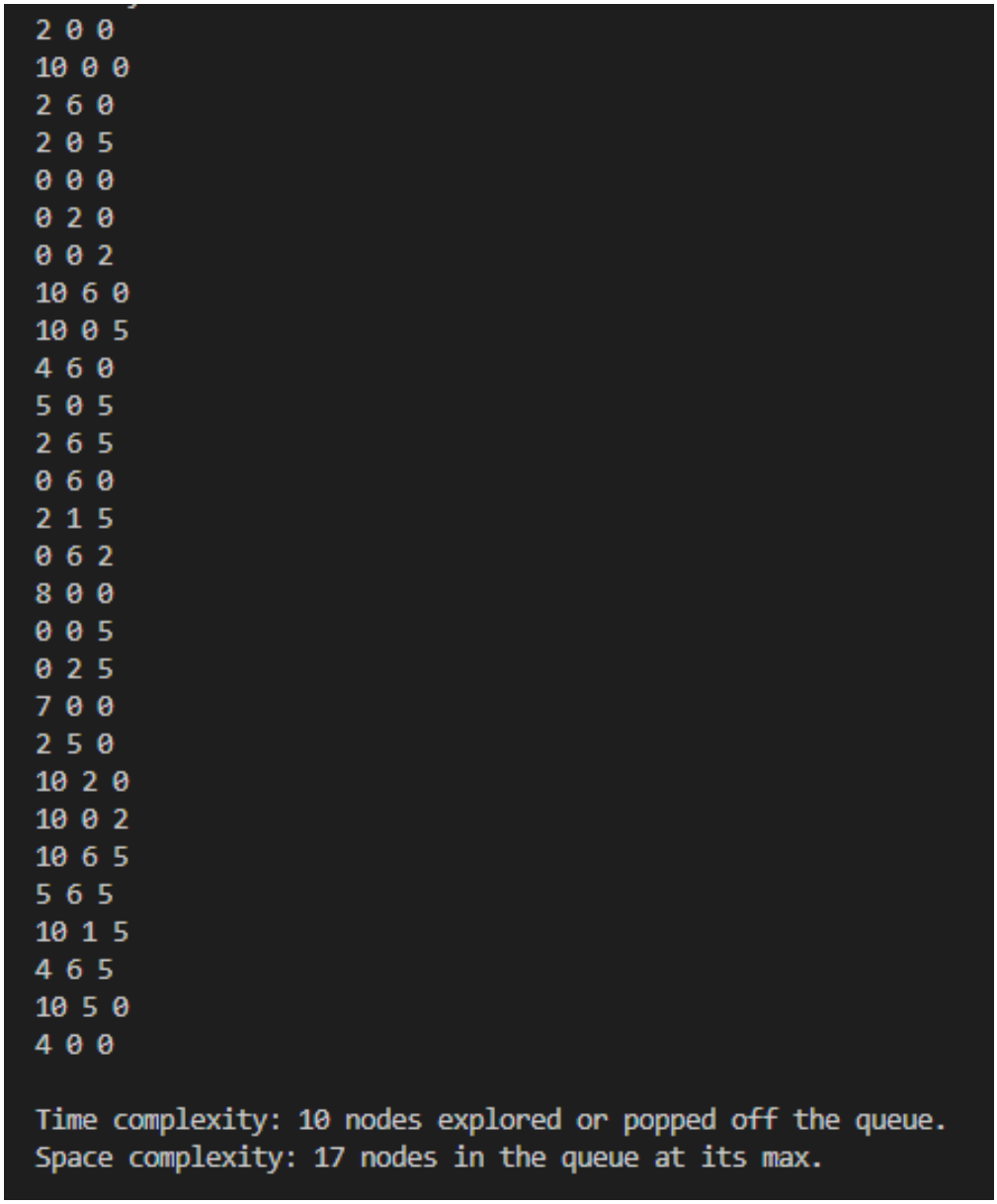


Figure 14: Breadth first search

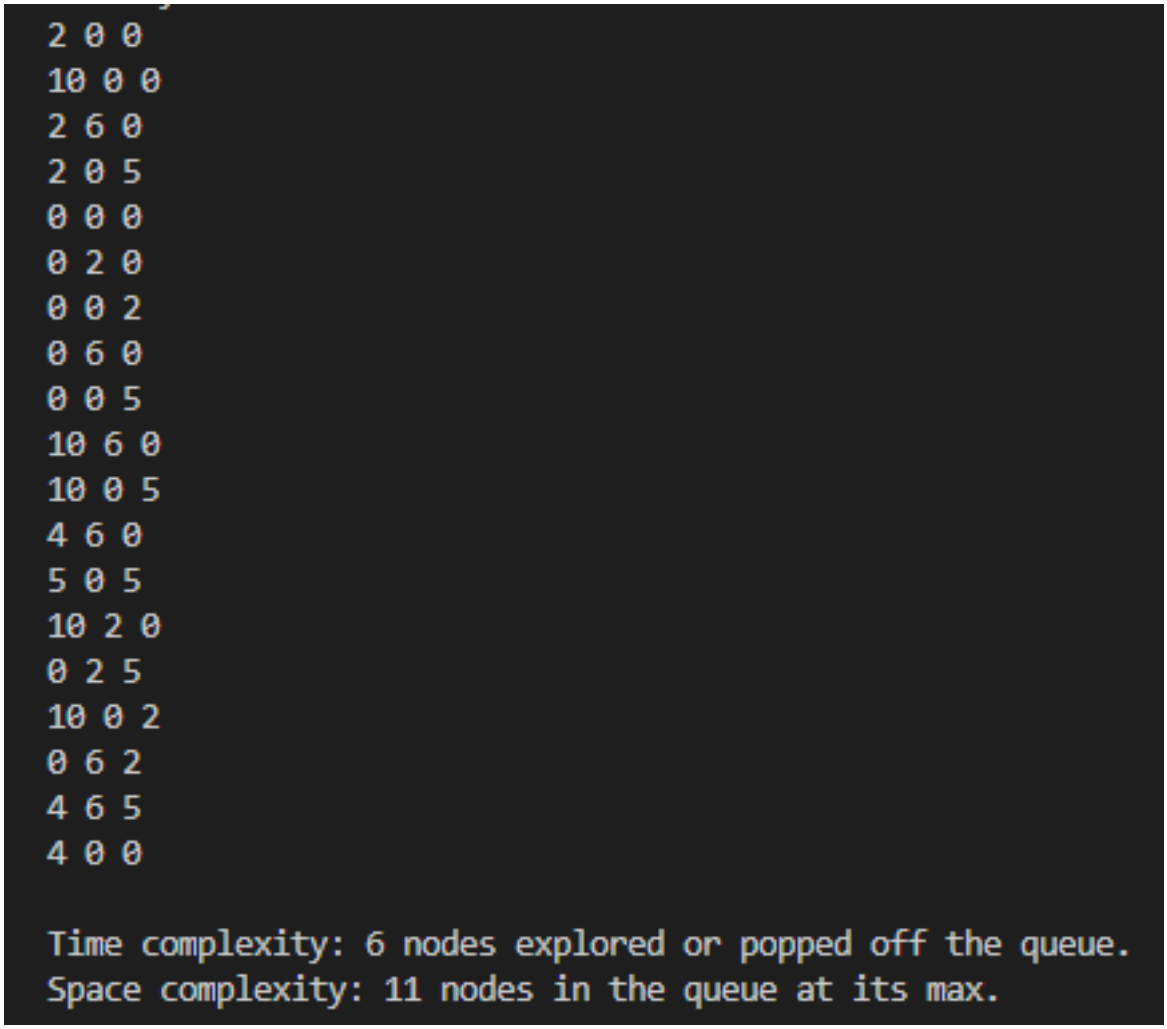


Figure 15: A search algorithm*

c) Start: *b1 = 3, b2 = 0, b3 = 0*

End: *b1 = 7, b2 = 0, b3 = 0*

```
3 0 0
0 0 0
0 3 0
0 0 3
3 6 0
3 0 5
0 0 5
0 3 5
3 6 5
8 0 0
3 5 0
0 5 0
2 6 0
0 5 3
3 5 5
0 5 5
2 6 5
8 0 5
8 5 0
3 6 4
0 6 4
3 0 4
9 0 4
7 6 0
0 6 0
7 0 0

Time complexity: 6 nodes explored or popped off the queue.
Space complexity: 18 nodes in the queue at its max.
```

Figure 16: Depth First Search

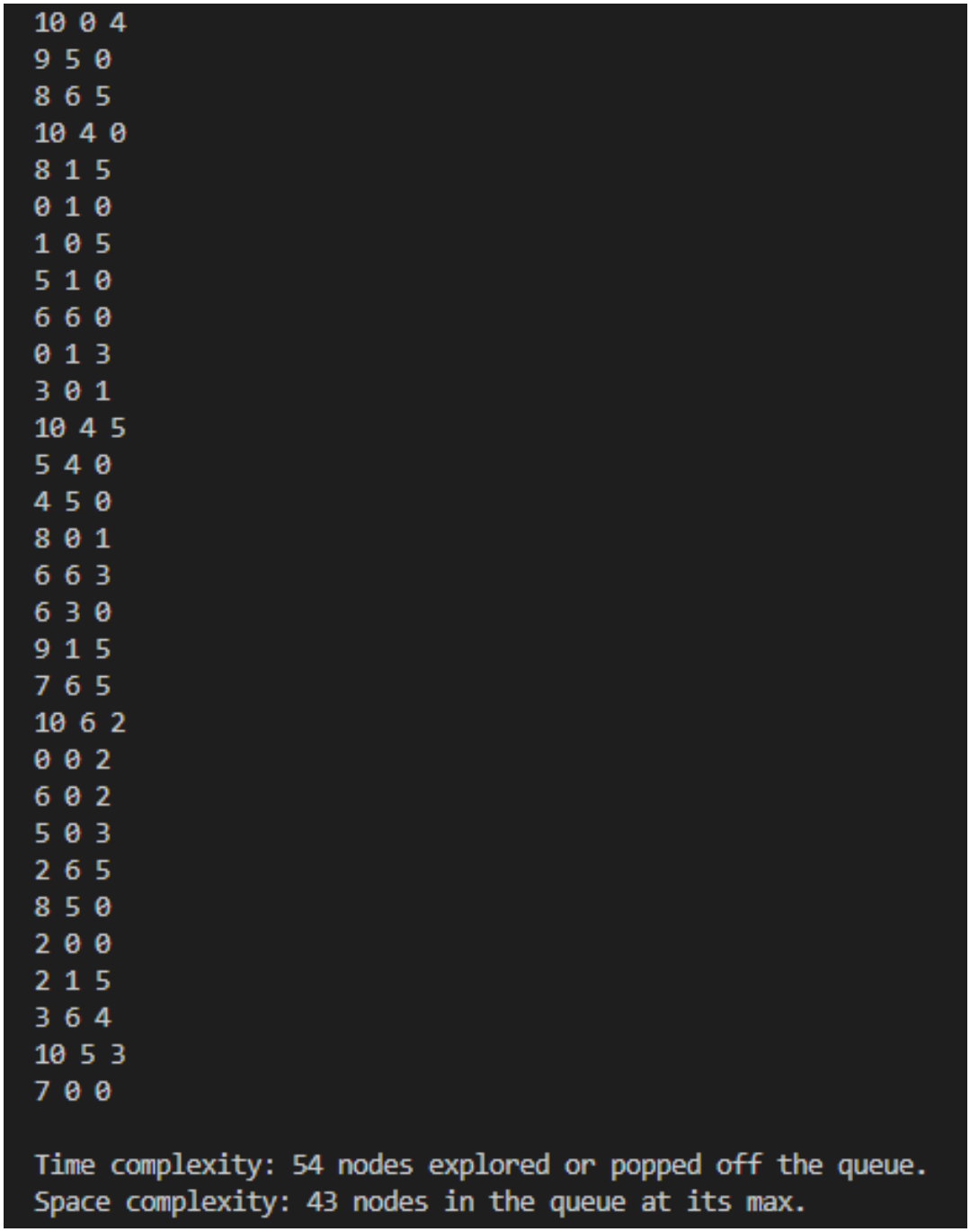


Figure 17: Breadth first search

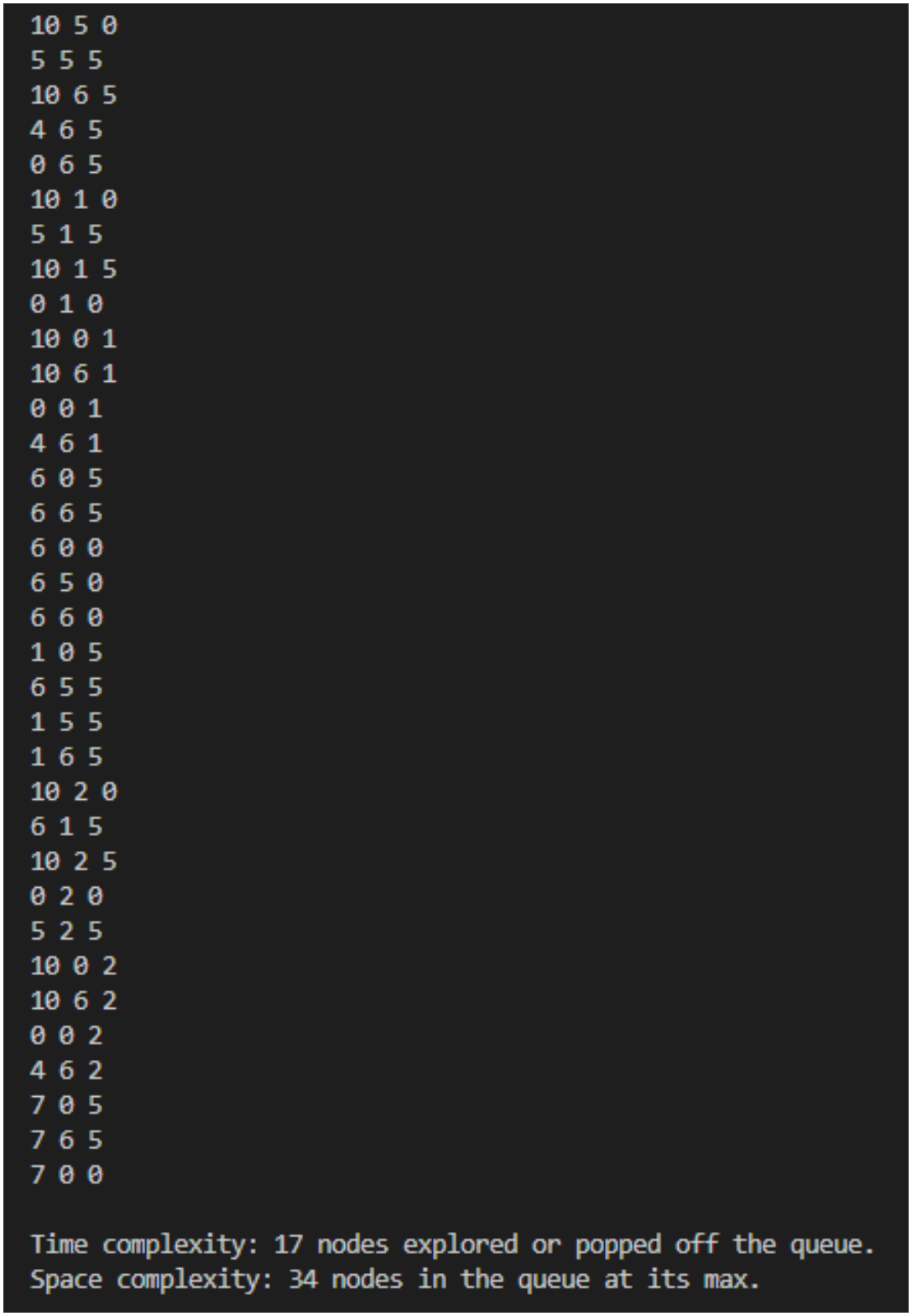


Figure 18: A search algorithm*

d) Start: *b1 = 3, b2 = 0, b3 = 0*

End: *b1 = 7, b2 = 0, b3 = 0*

*With a maximum capacity of [b1: 11, b2: 7, b3: 4].

```
7 7 3
7 0 0
7 3 0
0 3 0
3 3 4
7 3 4
0 3 4
7 0 4
7 4 0
4 7 0
3 4 4
3 0 4
3 4 0
3 7 1
3 0 1
0 3 1
3 0 0
4 0 0
3 1 0
0 1 3
3 1 4
4 0 4
7 1 0
3 5 0
1 7 0
0 5 3
8 0 0

Time complexity: 35 nodes explored or popped off the queue.
Space complexity: 66 nodes in the queue at its max.
```

Figure 19: Depth first search

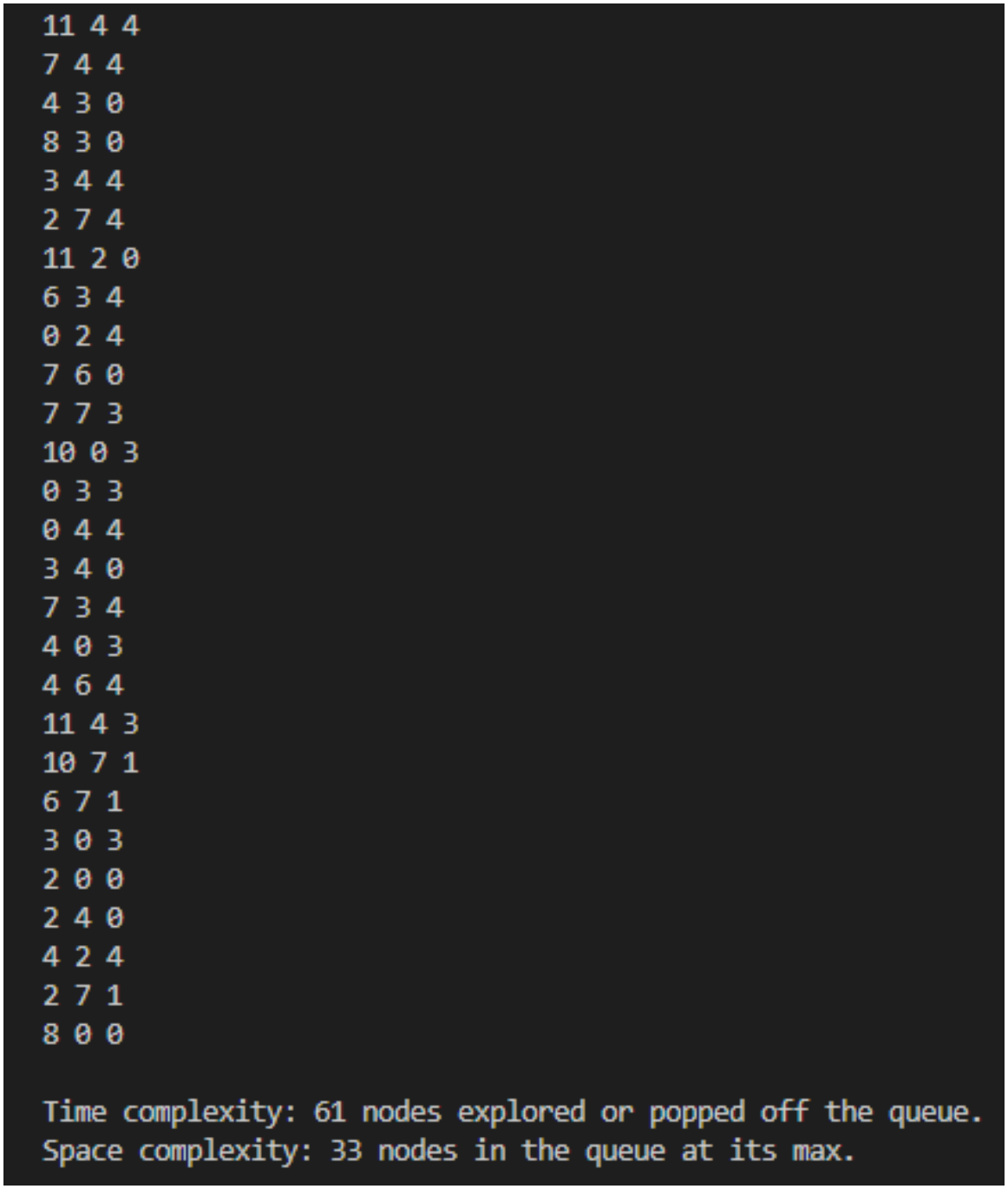


Figure 20: Breadth first search

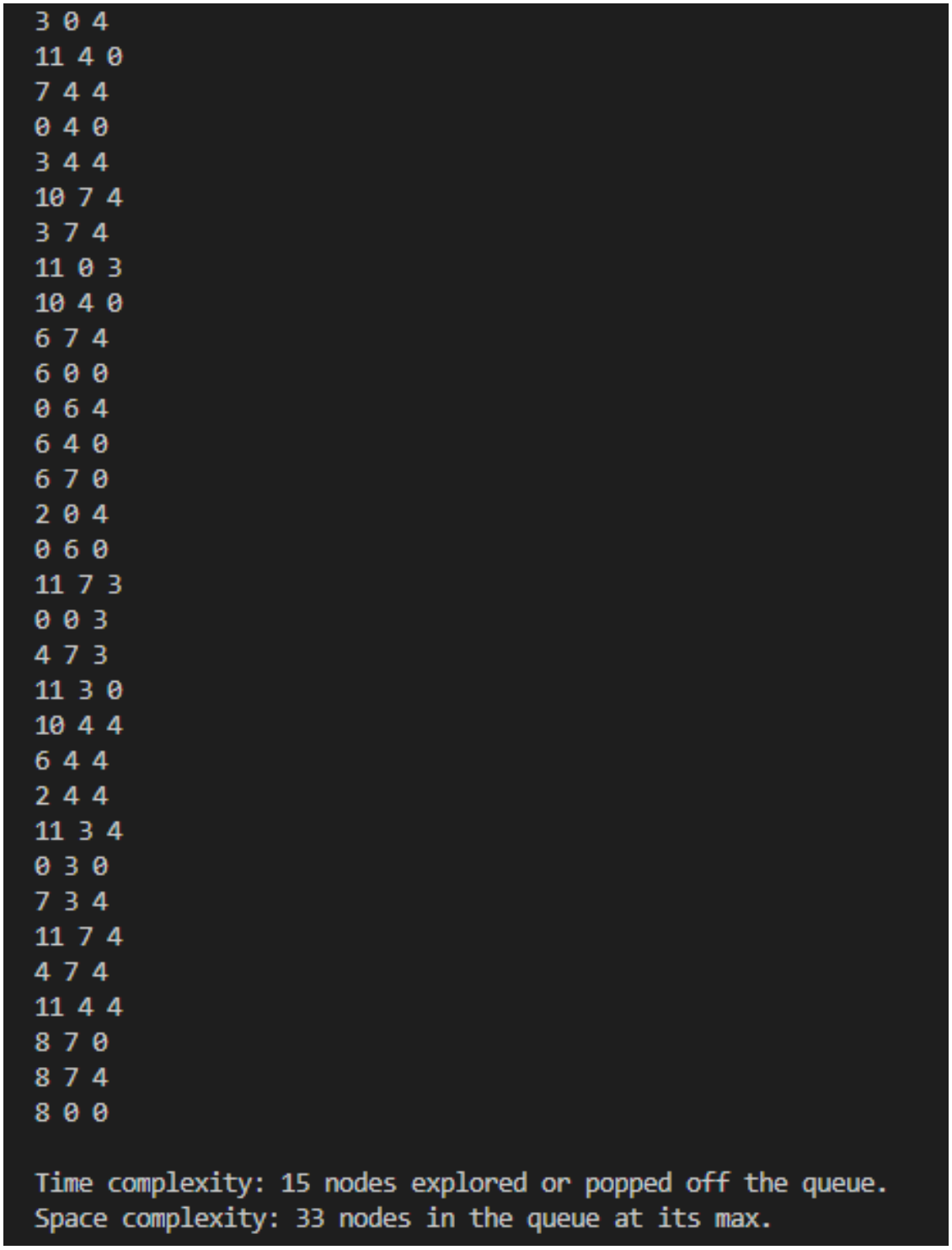


Figure 21: A search algorithm*

Observations

From the concluded data, it can be observed that the time complexity of the A* search algorithm's implementation has been lower in most cases as compared to the depth first and breadth first searches. Hence, the number of nodes to explore decrease the efficiency as the nodes increase and it takes longer to complete the search and achieve final state.

The time complexity when compared among the two uninformed searches, the depth first search turned out to be more efficient since its time complexity was lower in majority of the cases. Additionally, the total path cost is higher in depth first search since it has a higher number of nodes to be explored which makes breadth first search more optimal than depth first search.

Conclusively, a complete and optimal result with most efficiency is achieved using the informed A* search strategy.

Discussions

A Four Bottle Problem

Let us assume that there is another bottle added to the three-bottle problem. Increment of another bottle will increase the operator rules and the program will have more nodes to explore comparatively. The increase of nodes indicates that there will be more elements in the queue, and it will take longer to go through all of them increase the time and space complexity.

Water Overflow Dilemma

Let us assume that the water is not allowed to overflow in the three bottles. This will lead to the removal of 6 operators which would mean that more work would need to be done to achieve the goal state. More nodes will presumably be added to the queue which will lead to an increase in the time and space complexity.

Conclusion

After solving the problem using informed and uniformed strategies, it can be concluded that A* search algorithm is most efficient out of the three strategies used. It went through the least number of nodes to get the final state and its time and space complexity was lower than the other strategies.

Video Link

https://youtu.be/Mf4xnGBVZDM

Source Code

```
#
#Student name: Noor Ul Ain Khurshid
#Student id: 102763334
#

#max volumes of bottles
b1_max = 10
b2_max = 6
b3_max = 5

start_state = [10,0,0] #initial start state
final_state = [8,0,0] #final state

queue = [start_state] #initialize queue list and add initial state to queue
visited = [start_state] #initialize visited list and add initial state to
visited

#output solution
def output(num_visited, queue_cap):

    #prints all visited state
    for i in visited:
        print(i[0] , i[1], i[2])

    #prints if goal is reached
    if(i == final_state):
            print ('\nTime complexity:', str(num_visited),'nodes num_visited
or   off the queue.')
            print ('Space complexity:', str(queue_cap),'nodes in the queue at
its max.\n')

#breadth first search
def bfs():

    num_visited = 0 #counts number of nodes num_visited or
    queue_cap = 1 #checks max length of queue

    #loop while queue list in not empty
    while(queue):

        current = queue.pop(0) #remove first item from queue and save in
variable current
        num_visited += 1 #adds 1 to nodes num_visited or

        #current water level of bottle b1, b2 and b3
        x = current[0] #b1
```

```
        y = current[1] #b2
        z = current[2] #b3

        #calculated max length of queue
        if len(queue) > queue_cap:
                queue_cap = len(queue)

        #check if visited same as goal state
        if(visited[-1] == final_state):
            output(num_visited, queue_cap)
            return

        #fill b1:10 litre bottle
        if(x < b1_max and [b1_max, y, z] not in visited):
            visited.append([b1_max, y, z])
            queue.append([b1_max, y, z])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #fill b2:6 litre bottle
        if(y < b2_max and [x, b2_max, z] not in visited):
            visited.append([x, b2_max, z])
            queue.append([x, b2_max, z])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #fill b3:5 litre bottle
        if(z < b3_max and [x, y, b3_max] not in visited):
            visited.append([x, y, b3_max])
            queue.append([x, y, b3_max])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #empty b1:10 litre bottle
        if(x > 0 and [0, y, z] not in visited):
            visited.append([0, y, z])
            queue.append([0, y, z])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #empty b2:6 litre bottle
        if(y > 0 and [x, 0, z] not in visited):
            visited.append([x, 0, z])
            queue.append([x, 0, z])
```

```
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #empty b3:5 litre bottle
        if(z > 0 and [x, y, 0] not in visited):
            visited.append([x, y, 0])
            queue.append([x, y, 0])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b1:10 litre to b2:6 litre,    keep remaining in
b1:10 litre
        if(((x + y) >= b2_max and x > 0) and [x - (b2_max - y), b2_max, z] not
in visited):
            visited.append([x - (b2_max - y), b2_max, z])
            queue.append([x - (b2_max - y), b2_max, z])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b1:10 litre to b3:5 litre,    keep remaining in
b1:10 litre
        if(((x + z) >= b3_max and x > 0) and [x - (b3_max - z), y, b3_max] not
in visited):
            visited.append([x - (b3_max - z), y, b3_max])
            queue.append([x - (b3_max - z), y, b3_max])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b2:6 litre to b1:10 litre,    keep remaining in
b2:6 litre
        if(((y + x) >= b1_max and y > 0) and [b1_max, y - (b1_max - x), z] not
in visited):
            visited.append([b1_max, y - (b1_max - x), z])
            queue.append([b1_max, y - (b1_max - x), z])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b2:6 litre to b3:5 litre,     keep remaining in
b2:6 litre
        if(((y + z) >= b3_max and y > 0) and [x, y - (b3_max - z), b3_max] not
in visited):
            visited.append([x, y - (b3_max - z), b3_max])
            queue.append([x, y - (b3_max - z), b3_max])
```

```
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b3:5 litre to b1:10 litre,    keep remaining in
b3:5 litre
        if(((z + x) >= b1_max and z > 0) and [b1_max, y, z - (b1_max - x)] not
in visited):
            visited.append([b1_max, y, z - (b1_max - x)])
            queue.append([b1_max, y, z - (b1_max - x)])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b3:5 litre to b2:6 litre,     keep remaining in
b3:5 litre
        if(((z + y) >= b2_max and z > 0) and [x, b2_max, z - (b2_max - y)] not
in visited):
            visited.append([x, b2_max, z - (b2_max - y)])
            queue.append([x, b2_max, z - (b2_max - y)])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b1:10 litre to b2:6 litre,    no remaining
        if(((x + y) <= b2_max and x >= 0) and [0, (x + y), z] not in visited):
            visited.append([0, (x + y), z])
            queue.append([0, (x + y), z])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b1:10 litre to b3:5 litre,    no remaining
        if(((x + z) <= b3_max and x >= 0) and [0, y, (x + z)] not in visited):
            visited.append([0, y, (x + z)])
            queue.append([0, y, (x + z)])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b2:6 litre to b1:10 litre,    no remaining
        if(((y + x) <= b1_max and y >= 0) and [(x + y), 0, z] not in visited):
            visited.append([(x + y), 0, z])
            queue.append([(x + y), 0, z])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return
```

```
        #transfer water from b2:6 litre to b3:5 litre,     no remaining
        if(((y + z) <= b3_max and y >= 0) and [x, 0, (y + z)] not in visited):
            visited.append([x, 0, (y + z)])
            queue.append([x, 0, (y + z)])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b3:5 litre to b1:10 litre,    no remaining
        if(((z + x) <= b1_max and z >= 0) and [(x + z) , y, 0] not in
visited):
            visited.append([(x + z) , y, 0])
            queue.append([(x + z) , y, 0])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b3:5 litre to b2:6 litre,     no remaining
        if(((z + y) <= b2_max and z >= 0) and [x, (y + z), 0] not in visited):
            visited.append([x, (y + z), 0])
            queue.append([x, (y + z), 0])
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

#depth first search
def dfs():

    num_visited = 0 #counts number of nodes num_visited or
    queue_cap = 1 #checks max length of queue

    #loop while queue list in not empty
    while(queue):

        current = queue.pop(-1) #remove last item from queue and save in
variable current
        num_visited += 1 #adds 1 to nodes num_visited or

        #current water level of bottle b1, b2 and b3
        x = current[0] #b1
        y = current[1] #b2
        z = current[2] #b3

        #calculated max length of queue
        if len(queue) > queue_cap:
            queue_cap = len(queue)

        #check if visited same as goal state
```

```
        if(visited[-1] == final_state):
            output(num_visited, queue_cap)
            return

        #empty b1:10 litre bottle to...
        if(x > 0):

            #empty b1:10 litre bottle
            if([0, y, z] not in visited):
                visited.append([0, y, z])
                queue.append([0, y, z])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b1:10 litre to b2:6 litre,    keep remaining
in b1:10 litre
            if(((x + y) >= b2_max) and [x - (b2_max - y), b2_max, z] not in
visited):
                visited.append([x - (b2_max - y), b2_max, z])
                queue.append([x - (b2_max - y), b2_max, z])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b1:10 litre to b2:6 litre,    no remaining
            if(((x + y) <= b2_max) and [0, (x + y), z] not in visited):
                visited.append([0, (x + y), z])
                queue.append([0, (x + y), z])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b1:10 litre to b3:5 litre,    keep remaining
in b1:10 litre
            if(((x + z) >= b3_max) and [x - (b3_max - z), y, b3_max] not in
visited):
                visited.append([x - (b3_max - z), y, b3_max])
                queue.append([x - (b3_max - z), y, b3_max])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b1:10 litre to b3:5 litre,    no remaining
            if(((x + z) <= b3_max) and [0, y, (x + z)] not in visited):
                visited.append([0, y, (x + z)])
                queue.append([0, y, (x + z)])
                if(visited[-1] == final_state):
```

```
                    output(num_visited, queue_cap)
                    return

        #fill b1:10 litre bottle
        elif(x < b1_max):
            if([b1_max, y, z] not in visited):
                visited.append([b1_max, y, z])
                queue.append([b1_max, y, z])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

        #empty b2:6 litre bottle to...
        if(y > 0):

            #empty b2:6 litre bottle
            if([x, 0, z] not in visited):
                visited.append([x, 0, z])
                queue.append([x, 0, z])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b2:6 litre to b1:10 litre,    keep remaining
in b2:6 litre
            if(((y + x) >= b1_max) and [b1_max, y - (b1_max - x), z] not in
visited):
                visited.append([b1_max, y - (b1_max - x), z])
                queue.append([b1_max, y - (b1_max - x), z])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b2:6 litre to b1:10 litre,    no remaining
            if(((y + x) <= b1_max) and [(x + y), 0, z] not in visited):
                visited.append([(x + y), 0, z])
                queue.append([(x + y), 0, z])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b2:6 litre to b3:5 litre,     keep remaining
in b2:6 litre
            if(((y + z) >= b3_max) and [x, y - (b3_max - z), b3_max] not in
visited):
                visited.append([x, y - (b3_max - z), b3_max])
                queue.append([x, y - (b3_max - z), b3_max])
                if(visited[-1] == final_state):
```

```
                    output(num_visited, queue_cap)
                    return

            #transfer water from b2:6 litre to b3:5 litre,     no remaining
            if(((y + z) <= b3_max) and [x, 0, (y + z)] not in visited):
                visited.append([x, 0, (y + z)])
                queue.append([x, 0, (y + z)])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

        #fill b2:6 litre bottle
        elif(y < b2_max):
            if([x, b2_max, z] not in visited):
                visited.append([x, b2_max, z])
                queue.append([x, b2_max, z])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

        #empty b3:5 litre bottle to...
        if(z > 0):

            #empty b3:5 litre bottle
            if([x, y, 0] not in visited):
                visited.append([x, y, 0])
                queue.append([x, y, 0])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b3:5 litre to b1:10 litre,    keep remaining
in b3:5 litre
            if(((z + x) >= b1_max) and [b1_max, y, z - (b1_max - x)] not in
visited):
                visited.append([b1_max, y, z - (b1_max - x)])
                queue.append([b1_max, y, z - (b1_max - x)])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b3:5 litre to b1:10 litre,    no remaining
            if(((z + x) <= b1_max) and [(x + z) , y, 0] not in visited):
                visited.append([(x + z) , y, 0])
                queue.append([(x + z) , y, 0])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return
```

```
            #transfer water from b3:5 litre to b2:6 litre,     keep remaining
in b3:5 litre
            if(((z + y) >= b2_max) and [x, b2_max, z - (b2_max - y)] not in
visited):
                visited.append([x, b2_max, z - (b2_max - y)])
                queue.append([x, b2_max, z - (b2_max - y)])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

            #transfer water from b3:5 litre to b2:6 litre,     no remaining
            if(((z + y) <= b2_max) and [x, (y + z), 0] not in visited):
                visited.append([x, (y + z), 0])
                queue.append([x, (y + z), 0])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

        #fill b3:5 litre bottle
        elif(z < b3_max):
            if([x, y, b3_max] not in visited):
                visited.append([x, y, b3_max])
                queue.append([x, y, b3_max])
                if(visited[-1] == final_state):
                    output(num_visited, queue_cap)
                    return

#calculates path cost
def cost(current_path):
    path_cost = abs(current_path[0] - final_state[0]) + abs(current_path[1] -
final_state[1]) + abs(current_path[2] - final_state[2])
    return path_cost

#a* search
def astar():

    priority_queue = [[start_state, cost(start_state)]] #initialize priority
queue and set a cost

    num_visited = 0 #counts number of nodes num_visited or
    queue_cap = 1 #checks max length of queue

    #loop while priority queue list in not empty
    while priority_queue:

        priority_queue = sorted(priority_queue, key = lambda x: x[1])
#priority queue sorted by ascending order
```

```
        current = priority_queue.pop(0)[0] #remove last item from priority
queue and save only current state in variable current

        num_visited += 1 #adds 1 to nodes num_visited or

        #calculated max length of priority queue
        if len(priority_queue) > queue_cap:
                queue_cap = len(priority_queue)

        #current water level of bottle b1, b2 and b3
        x = current[0] #b1
        y = current[1] #b2
        z = current[2] #b3

        #check if visited same as goal state
        if(visited[-1] == final_state):
            output(num_visited, queue_cap)
            return

        #fill b1:10 litre bottle
        if(x < b1_max and [b1_max, y, z] not in visited):
            visited.append([b1_max, y, z])
            priority_queue.append(([b1_max, y, z], cost([b1_max, y, z])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #fill b2:6 litre bottle
        if(y < b2_max and [x, b2_max, z] not in visited):
            visited.append([x, b2_max, z])
            priority_queue.append(([x, b2_max, z], cost([x, b2_max, z])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #fill b3:5 litre bottle
        if(z < b3_max and [x, y, b3_max] not in visited):
            visited.append([x, y, b3_max])
            priority_queue.append(([x, y, b3_max], cost([x, y, b3_max])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #empty b1:10 litre bottle
        if(x > 0 and [0, y, z] not in visited):
            visited.append([0, y, z])
            priority_queue.append(([0, y, z], cost([0, y, z])))
```

```
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #empty b2:6 litre bottle
        if(y > 0 and [x, 0, z] not in visited):
            visited.append([x, 0, z])
            priority_queue.append(([x, 0, z], cost([x, 0, z])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #empty b3:5 litre bottle
        if(z > 0 and [x, y, 0] not in visited):
            visited.append([x, y, 0])
            priority_queue.append(([x, y, 0], cost([x, y, 0])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b1:10 litre to b2:6 litre,    keep remaining in
b1:10 litre
        if(((x + y) >= b2_max and x > 0) and [x - (b2_max - y), b2_max, z] not
in visited):
            visited.append([x - (b2_max - y), b2_max, z])
            priority_queue.append(([x - (b2_max - y), b2_max, z], cost([x -
(b2_max - y), b2_max, z])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b1:10 litre to b3:5 litre,    keep remaining in
b1:10 litre
        if(((x + z) >= b3_max and x > 0) and [x - (b3_max - z), y, b3_max] not
in visited):
            visited.append([x - (b3_max - z), y, b3_max])
            priority_queue.append(([x - (b3_max - z), y, b3_max], cost([x -
(b3_max - z), y, b3_max])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b2:6 litre to b1:10 litre,    keep remaining in
b2:6 litre
        if(((y + x) >= b1_max and y > 0) and [b1_max, y - (b1_max - x), z] not
in visited):
            visited.append([b1_max, y - (b1_max - x), z])
```

```
            priority_queue.append(([b1_max, y - (b1_max - x), z],
cost([b1_max, y - (b1_max - x), z])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b2:6 litre to b3:5 litre,     keep remaining in
b2:6 litre
        if(((y + z) >= b3_max and y > 0) and [x, y - (b3_max - z), b3_max] not
in visited):
            visited.append([x, y - (b3_max - z), b3_max])
            priority_queue.append(([x, y - (b3_max - z), b3_max], cost([x, y -
(b3_max - z), b3_max])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b3:5 litre to b1:10 litre,    keep remaining in
b3:5 litre
        if(((z + x) >= b1_max and z > 0) and [b1_max, y, z - (b1_max - x)] not
in visited):
            visited.append([b1_max, y, z - (b1_max - x)])
            priority_queue.append(([b1_max, y, z - (b1_max - x)],
cost([b1_max, y, z - (b1_max - x)])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b3:5 litre to b2:6 litre,     keep remaining in
b3:5 litre
        if(((z + y) >= b2_max and z > 0) and [x, b2_max, z - (b2_max - y)] not
in visited):
            visited.append([x, b2_max, z - (b2_max - y)])
            priority_queue.append(([x, b2_max, z - (b2_max - y)], cost([x,
b2_max, z - (b2_max - y)])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b1:10 litre to b2:6 litre,    no remaining
        if(((x + y) <= b2_max and x >= 0) and [0, (x + y), z] not in visited):
            visited.append([0, (x + y), z])
            priority_queue.append(([0, (x + y), z], cost([0, (x + y), z])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b1:10 litre to b3:5 litre,    no remaining
```

```
        if(((x + z) <= b3_max and x >= 0) and [0, y, (x + z)] not in visited):
            visited.append([0, y, (x + z)])
            priority_queue.append(([0, y, (x + z)], cost([0, y, (x + z)])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b2:6 litre to b1:10 litre,    no remaining
        if(((y + x) <= b1_max and y >= 0) and [(x + y), 0, z] not in visited):
            visited.append([(x + y), 0, z])
            priority_queue.append(([(x + y), 0, z], cost([(x + y), 0, z])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b2:6 litre to b3:5 litre,     no remaining
        if(((y + z) <= b3_max and y >= 0) and [x, 0, (y + z)] not in visited):
            visited.append([x, 0, (y + z)])
            priority_queue.append(([x, 0, (y + z)], cost([x, 0, (y + z)])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b3:5 litre to b1:10 litre,    no remaining
        if(((z + x) <= b1_max and z >= 0) and [(x + z) , y, 0] not in
visited):
            visited.append([(x + z) , y, 0])
            priority_queue.append(([(x + z) , y, 0], cost([(x + z) , y, 0])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

        #transfer water from b3:5 litre to b2:6 litre,     no remaining
        if(((z + y) <= b2_max and z >= 0) and [x, (y + z), 0] not in visited):
            visited.append([x, (y + z), 0])
            priority_queue.append(([x, (y + z), 0], cost([x, (y + z), 0])))
            if(visited[-1] == final_state):
                output(num_visited, queue_cap)
                return

#main functions
def main():
    while True:
        print("[1] Breadth First Search\n")
        print("[2] Depth First Search\n")
        print("[3] A* Search\n")
        choice = int(input("Enter your choice: "))
```

```
        if choice == 1:
            bfs() #breadth first search
            break
        elif choice == 2:
            dfs() #depth first search
            break
        elif choice == 3:
            astar() #a* search
            break
        else:
            print("Please enter a valid input\n")

main()
```