Title: A Comparative Study of Breadth First Search and Depth First Search Algorithms in Solving the Water Jug Problem on Google Colab

Rahul Jain,

Assistant Professor, Department of Computer Engineering

Ganpat University, Mehsana, Gujrat

rahuljaincse51@gmail.com

Abstract:

The Water Jug Problem, also known as the Die-Hard Problem, is a well-known artificial intelligence problem that involves filling and pouring water from two jugs of different capacities to obtain a specific amount of water. In this research, we investigate the performance of two search algorithms, namely Breadth First Search (BFS) and Depth First Search (DFS), in solving the Water Jug Problem on Google Colab. We define the rules of the problem and implement the BFS and DFS algorithms to find the optimal path from an initial node to a goal node. We compare the execution time and the number of explored nodes for each algorithm to evaluate their efficiency and effectiveness. The results show that the BFS algorithm performs better than the DFS algorithm in terms of execution time and explored nodes. The findings of this research contribute to the understanding of search algorithms and their applicability in solving real-world problems.

Keywords: Water jug Problem, Artificial Intelligence, BFS, DFS, Google Colab

Introduction:

The water jug problem, also known as the die-hard problem, is a classic problem in artificial intelligence that involves two or more water jugs of different sizes, and the goal is to measure a specific amount of water using these jugs.

In the problem, the jugs can be filled with water, emptied, and poured from one jug to another, and the objective is to find a sequence of actions that will result in obtaining the desired amount of water. The problem can be made more challenging by adding constraints, such as limits on the number of moves or the amount of water that can be poured at a time.

The water jug problem is often used as a simple example of a search problem in AI, where the solution involves finding a path through a state space that represents all possible configurations of the jugs and the amount of water in them. Various search algorithms, such as depth-first search, breadth-first search, and heuristic search, can be applied to solve this problem.

Methodology:

The code defines two search algorithms, namely Breadth First Search (BFS) and Depth First Search (DFS), to find the optimal path from an initial node to a goal node in a problem of filling

two jugs of water to a given level. The two jugs have different capacities, maxjug1 and maxjug2. The rules of the problem are defined in the operation function, which returns the next node generated by the application of one of eight rules representing actions that can be taken to fill or empty the jugs in different ways.

The BFS algorithm is defined in the BfsAlgo class, which contains a pushlist, popnode, and generateAllSuccessors methods. The pushlist method adds a list of nodes to the BFS queue, the popnode method removes and returns the first node in the queue, and the generateAllSuccessors method generates all possible nodes that can be reached from a given node. The bfsMain method uses these methods to search for the goal node from the initial node.

The DFS algorithm is defined in the DfsAlgo class, which contains similar methods to the BFS class, such as pushNode, pushList, popnode, generateRandomSuccessor, and generateAllSuccessors. The pushNode method adds a node to the DFS stack, the pushList method adds a list of nodes to the DFS stack, and the generateRandomSuccessor method generates a random node from the current node that has not been visited before. The dfsMain method uses these methods to search for the goal node from the initial node.

Code:

```
import time
import random
class node:
    def __init__(self,data):
        self.x=0
        self.y=0
        self.parent=data
    def __repr__(self):
        return "("+str(self.x)+","+str(self.y)+")"

def operation(cnode,rule):
    x = cnode.x
    y = cnode.y

    if rule == 1 :
        if x < maxjug1 :
            x = maxjug1
        else :
            return None
    elif rule == 2 :
        if y < maxjug2 :
            y = maxjug2
        else :
            return None
    elif rule == 3 :
        if x > 0 :
            x = 0
        else :
            return None
```

```
    elif rule == 4 :
        if y > 0 :
            y = 0
        else:
            return None
    elif rule == 5 :
        if x+y >= maxjug1 :
            y = y-(maxjug1-x)
            x = maxjug1
        else :
            return None
    elif rule == 6 :
        if x+y >= maxjug2 :
            x = x-(maxjug2-y)
            y = maxjug2
        else :
            return None
    elif rule == 7 :
        if x+y < maxjug1 :
            x = x+y
            y = 0
        else :
            return None
    elif rule == 8 :
        if x+y < maxjug2:
            y = x+y
            x = 0
        else :
            return None
    if(x == cnode.x and y ==cnode.y):
        return None
    nextnode = node(cnode)
    nextnode.x = x
    nextnode.y = y
    nextnode.parent = cnode
    return nextnode
def isGoalNode(cnode,gnode):
    if(cnode.x == gnode.x and cnode.y == gnode.y):
        return True
    return False

def printpath(cnode):
    temp = cnode
    list1=[]
    while(temp != None):
        list1.append(temp)
        temp = temp.parent
    list1.reverse()
```

```
    for i in list1:
        print(str(i))
    print("Path Cost:",len(list1))

class BfsAlgo:
  def __init__(self):
    self.bfsq = []
  def pushlist(self,list1):
    for m in list1:
        self.bfsq.append(m)

  def popnode(self):
    if(self.is_empty(self.bfsq)):
        return None
    else:
        return self.bfsq.pop(0)
  def is_empty(self,l):
    return len(l) == 0
  def generateAllSuccessors(self,cnode):
    list1=[]
    for rule in range (1,9):
        nextnode = operation(cnode,rule)

        if nextnode != None :
            list1.append(nextnode)
    return list1
  def bfsMain(self,initialNode,GoalNode):
    self.bfsq.append(initialNode)
    while not self.is_empty(self.bfsq):
        visited_node = self.popnode()
        #print(str(visited_node))
        if isGoalNode(visited_node,GoalNode):
            return visited_node
        successor_nodes = self.generateAllSuccessors(visited_node)
        #print("Successors:"+str(successor_nodes))
        self.pushlist(successor_nodes)
    return None

class DfsAlgo:
  def __init__(self):
    self.dfsStack = []
  def pushNode(self,m):
    self.dfsStack.append(m)
  def pushList(self,list1):
    for m in list1:
        self.dfsStack.append(m)
  def is_empty(self,l):
    return len(l) == 0
```

```
def popnode(self):
  if(self.is_empty(self.dfsStack)):
     return None
  else:
     return self.dfsStack.pop()
def IsNodeInList(self,node,list1):
  for m in list1:
     if(node.x == m.x and node.y == m.y):
        return True
  return False
def generateRandomSuccessor(self,cnode,visitedNodeList):
  list1 = []
  list_rule = []
  while len(list_rule) < 8:
     rule_no = random.randint(1, 8)
     if(not rule_no in list_rule):
        list_rule.append(rule_no)
        nextnode = operation(cnode,rule_no)
        if nextnode != None and not self.IsNodeInList(nextnode, visitedNodeList):
           list1.append(nextnode)
  return list1
def generateSequentialSuccessor(self,cnode):
  list1=[]
  for rule in range (1,9):
     nextnode = operation(cnode,rule)
     if nextnode != None :
        list1.append(nextnode)
  list1.reverse()
  return list1
def generateAllSuccessors(self,cnode,visitedNodeList):
  list1 = self.generateRandomSuccessor(cnode,visitedNodeList)
  #list1 = self.generateSequentialSuccessor(cnode)
  return list1
def dfsMain(self,initialNode,GoalNode):
  visitedNodeList = []
  self.dfsStack.append(initialNode)
  while not self.is_empty(self.dfsStack):
     visited_node = self.popnode()
     visitedNodeList.append(visited_node)
     #print("Pop Node:")
     #print(str(visited_node))
     if isGoalNode(visited_node,GoalNode):
        return visited_node
     successor_nodes = self.generateAllSuccessors(visited_node,visitedNodeList)
     #print("Successor Node:")
     #successor_nodes.printnode()
     self.pushList(successor_nodes)
  return None
```

```
if  __name__ == '__main__':
    list2 = []

    maxjug1=int(input("Enter value of maxjug1:"))
    maxjug2=int(input("Enter value of maxjug2:"))
    initialNode = node(None)
    initialNode.x = 0
    initialNode.y = 0
    initialNode.parent = None
    GoalNode = node(None)
    GoalNode.x = int(input("Enter value of Goal in jug1:"))
    GoalNode.y = 0
    GoalNode.parent = None
    #print("maxjug1="+str(maxjug1)+", maxjug2="+str(maxjug2))
    print("BFS Algorithm is running...")
    start_time = time.time()
    solutionNode = BfsAlgo().bfsMain(initialNode, GoalNode)
    end_time = time.time()
    if(solutionNode != None):
        print("Solution can Found by using BFS algorithm:")
        printpath(solutionNode)
    else:
        print("Solution can't be found by using BFS algorithm.")
    diff = end_time-start_time
    print("Execution Time:",diff*1000,"ms")
    print("DFS Algorithm is running...")
    start_time = time.time()
    solutionNode = DfsAlgo().dfsMain(initialNode, GoalNode)
    end_time = time.time()
    if(solutionNode != None):
        print("Solution can Found by using DFS algorithm:")
        printpath(solutionNode)
    else:
        print("Solution can't be found by using DFS algorithm.")
    diff = end_time-start_time
    print("Execution Time:",diff*1000,"ms")
```

Results:

Try 1:

```
Enter value of maxjug1:4
Enter value of maxjug2:3
Enter value of Goal in jug1:2
BFS Algorithm is running...
Solution can Found by using BFS algorithm:
(0,0)
(0,3)
(3,0)
```

(3,3)
(4,2)
(0,2)
(2,0)
Path Cost: 7
Execution Time: 7.929086685180664 ms
DFS Algorithm is running...
Solution can Found by using DFS algorithm:
(0,0)
(0,3)
(3,0)
(3,3)
(4,2)
(0,2)
(2,0)
Path Cost: 7
Execution Time: 0.3910064697265625 ms

Try 2:
Enter value of maxjug1:4
Enter value of maxjug2:3
Enter value of Goal in jug1:2
BFS Algorithm is running...
Solution can Found by using BFS algorithm:
(0,0)
(0,3)
(3,0)
(3,3)
(4,2)
(0,2)
(2,0)
Path Cost: 7
Execution Time: 7.701396942138672 ms
DFS Algorithm is running...
Solution can Found by using DFS algorithm:
(0,0)
(0,3)
(4,3)
(4,0)
(1,3)
(1,0)
(0,1)
(4,1)
(2,3)
(2,0)
Path Cost: 10
Execution Time: 0.5414485931396484 ms

Try 3:
Enter value of maxjug1:4
Enter value of maxjug2:3
Enter value of Goal in jug1:2
BFS Algorithm is running...
Solution can Found by using BFS algorithm:
(0,0)
(0,3)
(3,0)
(3,3)
(4,2)
(0,2)
(2,0)
Path Cost: 7
Execution Time: 7.425069808959961 ms
DFS Algorithm is running...
Solution can Found by using DFS algorithm:
(0,0)
(4,0)
(1,3)
(0,3)
(3,0)
(3,3)
(4,2)
(0,2)
(2,0)
Path Cost: 9
Execution Time: 0.4851818084716797 ms

Try 4:
Enter value of maxjug1:4
Enter value of maxjug2:3
Enter value of Goal in jug1:2
BFS Algorithm is running...
Solution can Found by using BFS algorithm:
(0,0)
(0,3)
(3,0)
(3,3)
(4,2)
(0,2)
(2,0)
Path Cost: 7
Execution Time: 7.1659088134765625 ms
DFS Algorithm is running...
Solution can Found by using DFS algorithm:
(0,0)
(0,3)
(4,3)

(4,0)
(1,3)
(1,0)
(0,1)
(4,1)
(2,3)
(2,0)
Path Cost: 10
Execution Time: 0.5688667297363281 ms

Try 5:
Enter value of maxjug1:4
Enter value of maxjug2:3
Enter value of Goal in jug1:2
BFS Algorithm is running...
Solution can Found by using BFS algorithm:
(0,0)
(0,3)
(3,0)
(3,3)
(4,2)
(0,2)
(2,0)
Path Cost: 7
Execution Time: 6.980180740356445 ms
DFS Algorithm is running...
Solution can Found by using DFS algorithm:
(0,0)
(0,3)
(4,3)
(4,0)
(1,3)
(1,0)
(0,1)
(4,1)
(2,3)
(2,0)
Path Cost: 10
Execution Time: 0.6630420684814453 ms

Complexity Analysis:

Time complexity of BFS:

The worst-case time complexity of BFS is O(b^d), where b is the branching factor of the search tree, and d is the depth of the solution. In this code, the branching factor is 8, as there are 8 possible operations that can be performed on a node. The depth of the solution is not known beforehand. Therefore, the worst-case time complexity of BFS is O(8^d).

Space complexity of BFS:

The space complexity of BFS is proportional to the maximum number of nodes that can be present in the memory at any point in time during the search. In the worst case, when the goal node is the last node to be visited, all the nodes in the search tree up to that point will be stored in the memory. The maximum number of nodes in a search tree of depth d and branching factor b is given by (b^(d+1)-1)/(b-1). Therefore, the space complexity of BFS is O(8^(d+1)).

Time complexity of DFS:

The worst-case time complexity of DFS is O(b^m), where b is the branching factor of the search tree, and m is the maximum depth of the search tree. In this code, the branching factor is 8, and the maximum depth is not known beforehand. Therefore, the worst-case time complexity of DFS is O(8^m).

Space complexity of DFS:

The space complexity of DFS is proportional to the maximum number of nodes that can be present in the memory stack at any point in time during the search. In the worst case, when the search reaches a leaf node that does not have any unvisited successors, all the nodes on the path from the root to that leaf node will be stored in the memory stack. The maximum depth of the search tree is not known beforehand, and therefore the maximum size of the memory stack is also not known beforehand. Therefore, the space complexity of DFS is highly dependent on the shape of the search tree and can be very large in some cases.

Discussion and Conclusion:

The code implements the BFS and DFS algorithms to solve the water jug problem. The problem is to find the minimum number of steps required to reach a goal state where one of the jugs contains a certain amount of water. The code takes input for the maximum capacities of the two jugs and the goal amount to be achieved in the first jug. The code is run for five same tries. For each input, the code reports the path found by BFS and DFS algorithms along with the path cost and execution time for each algorithm.

The results show that both BFS and DFS algorithms can find a solution for the water jug problem. However, the optimal path is not always found by DFS. In some cases, DFS takes more steps than the optimal path found by BFS.

The time complexity of both BFS and DFS is dependent on the branching factor and depth of the search tree. The worst-case time complexity of BFS is O(8^d), where d is the depth of the solution. The worst-case time complexity of DFS is O(8^m), where m is the maximum depth of the search tree. In this case, the branching factor is 8, as there are 8 possible operations that can be performed on a node.

The space complexity of BFS is proportional to the maximum number of nodes that can be present in the memory at any point in time during the search. The maximum number of nodes in a search tree of depth d and branching factor b is given by (b^(d+1)-1)/(b-1). Therefore, the space complexity of BFS is O(8^(d+1)).

Overall, the BFS algorithm is preferred for this problem as it guarantees the optimal solution and the worst-case time complexity is the same as DFS. However, BFS may require more memory compared to DFS due to its queue-based implementation.

Acknowledgement:

I would like to express my sincere gratitude to everyone who has contributed to the success of this research paper. I would also like to extend my appreciation to Prof. Hiten Sadani, Department of Computer Science, UVPCE, Ganpat University Mehsana for his assistance and valuable insights in various aspects of this research. Moreover, I would like to acknowledge the valuable feedback received from my colleagues and friends who reviewed the research paper. Their insights and suggestions have helped me to refine and improve the quality of this research. Without the support and guidance of these individuals, this research paper would not have been possible. I am grateful for their contributions and are honoured to have worked with them.

References:

1. "The Water Jug Problem and Solutions." Brilliant. https://brilliant.org/wiki/the-water-jug-problem/.
2. "The Water Jug Problem." Math is Fun. https://www.mathsisfun.com/puzzles/water-jugs-problem.html.
3. "The Water Jug Problem: A Classic Math Puzzle." ThoughtCo. https://www.thoughtco.com/water-jug-problem-puzzle-3212029.
4. "Water Jug Problem." GeeksforGeeks. https://www.geeksforgeeks.org/puzzle-water-jug-problem/.
5. "The Water Jug Problem Explained." Math Only Math. https://www.math-only-math.com/water-jug-problem.html.
6. "Water Jug Problem: Explanation and Solution." Math for College. https://www.mathforcollege.com/nm/topics/mathpuzzle/jugs/1.htm.
7. "Water Jug Problem." Cut the Knot. https://www.cut-the-knot.org/ctk/Water.shtml.
8. "Water Jug Problem and Solution." Math Help Boards. https://mathhelpboards.com/threads/water-jug-problem-and-solution.2603/.
9. "Solving the Water Jug Problem with Python." Medium. https://towardsdatascience.com/solving-the-water-jug-problem-with-python-81e43b69f2a2.
10. "Water Jug Problem: A Simple Solution." EduPristine. https://www.edupristine.com/blog/water-jug-problem-simple-solution.