21BCE7371
RADHA KRISHNA GARG

AI LAB ASSIGNMENT-4

1. DEPTH FIRST SEARCH [DFS]

In the depth-first search (DFS), we go as far down as possible into search tree / graph before backing up and trying alternatives. It works by always generating a descendent of the most recently expanded node until some depth cut off is reached and then backtracks to next most recently expanded node and generates one of its descendants. DFS is memory efficient, as it only stores a single path from the root to leaf node along with the remaining unexpanded siblings for each node on the path.

We can implement DFS by using two lists called OPEN and CLOSED. The OPEN list contains those states that are to be expanded, and CLOSED list keeps track of states already expanded. Here OPEN and CLOSED lists are maintained as stacks. If we discover that first element of OPEN is the goal state then search terminates successfully. We can get track of the path through state space as we traversed, but in those situations where many nodes after expansion are in the closed list, we fail to keep track of our path. This information can be obtained by modifying CLOSED list by putting pointer back to its parent in the search tree.

Algorithm

Input: START and GOAL states.

Local variables: OPEN, CLOSED, ~~STATE-X~~ RECORD-X, SUCCs, FOUND;

Output: Yes or NO

Method:

- initialize OPEN list with START' nil and CLOSED = ϕ;
- FOUND = false;
- While (OPEN $\neq \phi$ and FOUND = false) do

  {

  - remove the first ~~state~~ record (initially (START, nil)) from OPEN and call it ~~STATE-X~~ RECORD-X;
  - put ~~STATE-X~~ RECORD-X in the front of CLOSED list {maintai-ned as stack};
  - If STATE-X of RECORD-X = GOAL then FOUND = true else

    {

    - perform EXPAND operation on STATE-X, producing a list of ~~SUCCs~~ records called SUCCS;
    - remove from successors those states, if any, that are in the CLOSED Lists:
    - ~~append~~ insert SUCCS ~~at~~ in the ~~End~~ front of the OPEN list:

    }

  } /* End While */

- if FOUND = true then return ~~Yes else return NO~~ the path by tracing through the pointers to the parents on the CLOSED list else return NO.
- Stop.

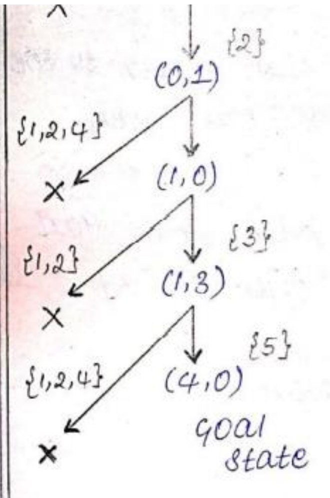
Let us see the search tree generation from start state of the water jug problem using DFS algorithm. ~~At each state, we apply first applicable~~

rule. if it ~~generates previously generated state then cross it and try another rule in the sequence to avoid the looping. If new state is generated then cross it and try another rule in sequence to avoid the looping. If new state is generated then expand this state in breadth first problem/ fashion.~~

~~The rules applied and enclosed for water jug problem are:~~

Let us see the search tree generation from start state of the water jug problem using DFS algorithm.

Water jug problem		
search tree generation using DFS	OPEN list	CLOSED list
Start State (0,0)	[((0,0), nil)]	
↓ {1} (5,0)	[((5,0), (0,0))]	[((0,0), nil)]
{2} → X ↓ {3} (5,3)	[((5,3), (5,0))]	[((5,0),(0,0)), ((0,0), nil)]
↓ {2} (0,3)	[((0,3), (5,3))]	[((5,3),(5,0)), ((5,0),(0,0)), ((0,0), (nil)]
{1,4} → X ↓ {5} (3,0)	[((3,0), (0,3))]	[((0,3),(5,3)), ((5,3),(5,0)), ((5,0),(0,0)), ((0,0), nil)]
↓ {3} (3,3)	[((3,3), (3,0))]	[((3,3),(3,0)), ((0,3),(5,3)), ((5,3),(5,0)), ((5,0),(0,0)), ((0,0), nil)]
{1,2,4} → X ↓ {7} (5,1)	⋮	⋮
↓ {2} (0,1)	⋮	⋮
{1,2,4} ↓	⋮	⋮



[(4,0), (1,3)]

[((4,0), (1,3)), ((1,3), (1,0)), ((1,0), (0,1)), ((0,1), (5,1)), ((5,1), (3,3)), ((3,3), (3,0)), ((3,0), (0,3)), ((0,3), (5,3)), ((5,3), (5,0)), ((5,0), (0,0)), ((0,0), nil)]

The path is obtained from the list stored in CLOSED. The solution path is

(0,0) → (5,0) → (5,3) → (0,3) → (3,0) → (3,3) → (5,1) → (0,1) → (1,0) → (1,3) → (4,0)

- BFS is effective when the search tree has a low branching factor.
- BFS can work even in trees that are infinitely deep.
- BFS requires a lot of memory as number of nodes in level of the tree increases exponentially.
- BFS is superior when the GOAL exists in the upper right portion of a search tree.
- BFS gives optimal solution.
- DFS is effective when there are few sub trees in the search tree that have only one connection point to the rest of states.
- DFS is best when the GOAL exists in the lower left portion of search tree.
- DFS can be dangerous when the path closer to the START and farther from the GOAL has been choosen.
- DFS is memory efficient as the path from start to current node is stored. Each node should contain state and its parent.
- DFS may not give optimal solution.

Algorithm:

Input: START and GOAL states

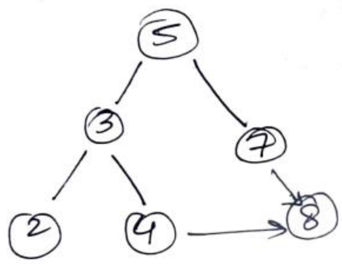
Local variables: OPEN, CLOSED, STATE-X, SUCCS, FOUND;

Output: yes or NO

Method:

- initialize OPEN list with START and CLOSED = ∅
- FOUND = false;

DFS



If we implement using DFS we travel to depth first and then to other nodes so it should be

Output :- 5 3 2 4 8 7

main.py

```python
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node):  #function for dfs
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

OUTPUT

Shell

```
Following is the Depth-First Search
5
3
2
4
8
7
>
```

2.BREADTH FIRST SEARCH [BFS]

Breadth first search:

The breadth first search expands all the states one step away from the start state and then expands all states two steps from start, then three steps, etc., until a goal state is reached. All successor states are examined at the same depth before going deeper. The BFS always gives an optimal path or solution.

This search is implemented using two lists called OPEN and CLOSED. The OPEN list contains those states that are to be expanded and CLOSED list keeps track of states already expanded. Here OPEN list is maintained as a queue and CLOSED list as a stack. for the sake of simplicity, we are writing BFS algorithm for checking whether a goal node exists or not. further, this algorithm can be modified to get a path from start to goal nodes by maintaining CLOSED list with pointer back to its parent in the search tree.

- While (OPEN ≠ $\emptyset$ and FOUND = false) do

{

- ~~perform EXPAND operation~~ remove the first state from OPEN and call it STATE-X;
- put STATE-X in the front of CLOSED list;
- if STATE-X = GOAL then FOUND = true else

{

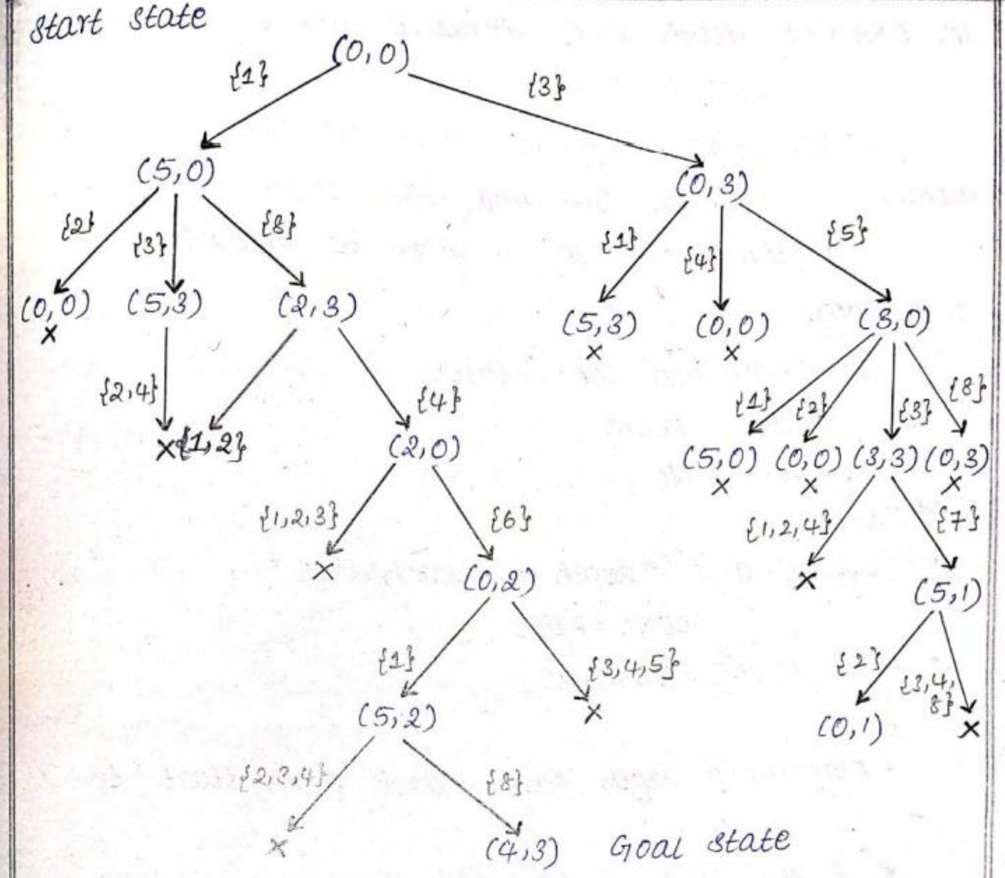
- perform EXPAND operation on STATE-X, producing a list of SUCCS;
- remove from sucessors those states, if any, that are in the CLOSED list;
- append SUCCS at the end of OPEN list;

}

} /* end while */

- if FOUND = true then return yes else return NO
- stop

Let us see the search tree generation from start state of the water jug problem using BFS algorithm. At each state, we apply first applicable rule. If it generates previously generated state then cross it and try another rule in the sequence to avoid the looping. If new state is generated then expand this state in breadth first fashion.



search tree generation using BFS

search tree is developed level wise. this is not memory efficient as partially developed tree is to be kept in the memory but it finds optimal solution or path. We can easily see the path from start to goal by tracing the tree from goal state to start state through parent link. this path is optimal and we cannot get a path shorter than is.

solution path : (0,0) → (5,0) → (2,3) → (2,0) → (0,2) → (5,2) → (4,3)

Q 2 Jugs A and B, Jug A can hold → 3 L
Jug B → 4L, There's a requirement to obtain 2 L
of water from one of the Jugs somehow

Sol Steps we can perform
1. Empty a Jug
2. fill a Jug
3. Pour water from one jug to another until one of
the Jugs is either empty or full.

USING BFS

States as (A,B)
A → Amount of water in A
B → Amount of water in B

(0,0) → initial
(0,2), (2,0) → final

Steps using this approach
1. ~~fill~~ Empty a Jug :→ Transition could be (A,B) → (0,B)
Let's assume we empty Jug A.
2. fill a Jug → (0,0) → (A,0), Assume filling Jug A.
3. Pour water → (A,B) → (A−d, B+d)

Implementation in Code

IMPLEMENTAIION IN CODE:

main.py

```python
from collections import deque

def BFS(a, b, target):

    # Map is used to store the states, every
    # state is hashed to binary value to
    # indicate either that state is visited
    # before or not
    m = {}
    isSolvable = False
    path = []

    # Queue to maintain states
    q = deque()

    # Initialing with initial state
    q.append((0, 0))

    while (len(q) > 0):

        # Current state
        u = q.popleft()

        # If this state is already visited
        if ((u[0], u[1]) in m):
            continue

        # Doesn't met jug constraints
        if ((u[0] > a or u[1] > b or
             u[0] < 0 or u[1] < 0)):
            continue

        # Filling the vector for constructing
        # the solution path
        path.append([u[0], u[1]])

```

main.py

```python
        path.append([u[0], u[1]])

        # Marking current state as visited
        m[(u[0], u[1])] = 1

        # If we reach solution state, put ans=1
        if (u[0] == target or u[1] == target):
            isSolvable = True

            if (u[0] == target):
                if (u[1] != 0):

                    # Fill final state
                    path.append([u[0], 0])
            else:
                if (u[0] != 0):

                    # Fill final state
                    path.append([0, u[1]])

            # Print the solution path
            sz = len(path)
            for i in range(sz):
                print("(", path[i][0], ",",
                           path[i][1], ")")
            break

        # If we have not reached final state
        # then, start developing intermediate
        # states to reach solution state
        q.append([u[0], b]) # Fill Jug B
        q.append([a, u[1]]) # Fill Jug A

        for ap in range(max(a, b) + 1):

            # Pour amount ap from Jug B to Jug A
```

main.py

```python

        for ap in range(max(a, b) + 1):

            # Pour amount ap from Jug B to Jug A
            c = u[0] + ap
            d = u[1] - ap

            # Check if this state is possible or not
            if (c == a or (d == 0 and d >= 0)):
                q.append([c, d])

            # Pour amount ap from Jug A to Jug B
            c = u[0] - ap
            d = u[1] + ap

            # Check if this state is possible or not
            if ((c == 0 and c >= 0) or d == b):
                q.append([c, d])

        # Empty Jug B
        q.append([a, 0])

        # Empty Jug A
        q.append([0, b])

    # No, solution exists if ans=0
    if (not isSolvable):
        print ("No solution")

# Driver code
if __name__ == '__main__':

    JugA, JugB, target = 3, 4, 2
    print("Path from initial state "
          "to solution state ::")

    BFS(JugA, JugB, target)
```

OUTPUT

```
Path from initial state to solution state ::
( 0 , 0 )
( 0 , 4 )
( 3 , 0 )
( 3 , 4 )
( 3 , 1 )
( 0 , 3 )
( 3 , 3 )
( 2 , 4 )
( 2 , 0 )
```

#Path from initial state to solution state ::

(0 , 0)

(0 , 4)

(3 , 0)

(3 , 4)

(3 , 1)

(0 , 3)

(3 , 3)

(2 , 4)

(2 , 0)