<u>Shri. Ramdeobaba College of Engineering & Management</u>

<u>Nagpur-13</u>

<u>Department of Computer Application</u>

Session: 2022-2023

Submission for

Course Name: Artificial Intelligence Lab

Course Code: MCP640

Name of the Student: Shiwani A. Gomase

Class Roll No: 27

Shift: II

Batch: 1

Under the Guidance of
Prof. Aparna Gurjar

Date of submission: 10/12/2023

INDEX

PRACTICAL NO: 1

AIM :- Implementing un-informed search strategies. Breadth First Search (BFS) and Goal based BFS

THEORY:- Breadth First Search or BFS for a Graph

The **Breadth First Search (BFS)** algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.

Relation between BFS for Graph and Tree traversal:

Breadth-First Traversal (or Search) for a graph is similar to the Breadth-First Traversal of a tree.
The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

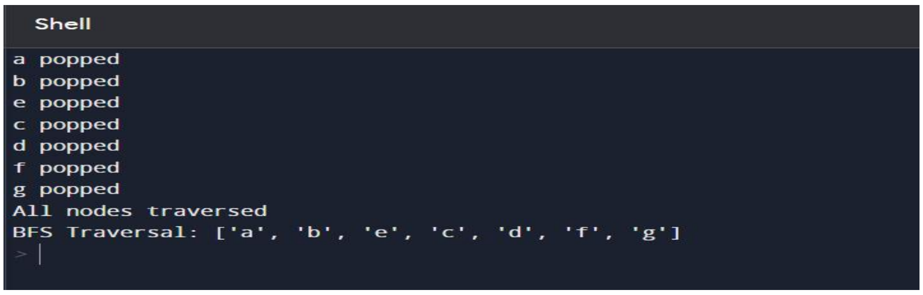
- Visited and
- Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a **queue data structure** for traversal.

SOURCE CODE:

```
def bfs(graph,root):
    visited, queue= [root], [root]
    while queue:
        vertex = queue.pop(0)
        print(vertex, "popped")
        if vertex in graph.keys():
            for w in graph[vertex]:
                if w not in visited:
                    visited.append(w)
                    queue.append(w)
    print("All nodes traversed")
    return (visited)
tree1={1:[2,3],2:[4,5],3:[6,7]}
graph2={'a':['b','e','c'],
        'b':['a','d','e'],
        'c':['a','f','g'],
        'd':['b','e'],
        'e':['a','b','d'],
        'f':['c'],
        'g':['c']}
v2=bfs(graph2, 'a')
print("BFS Traversal:", v2)
```

OUTPUT:



Shell

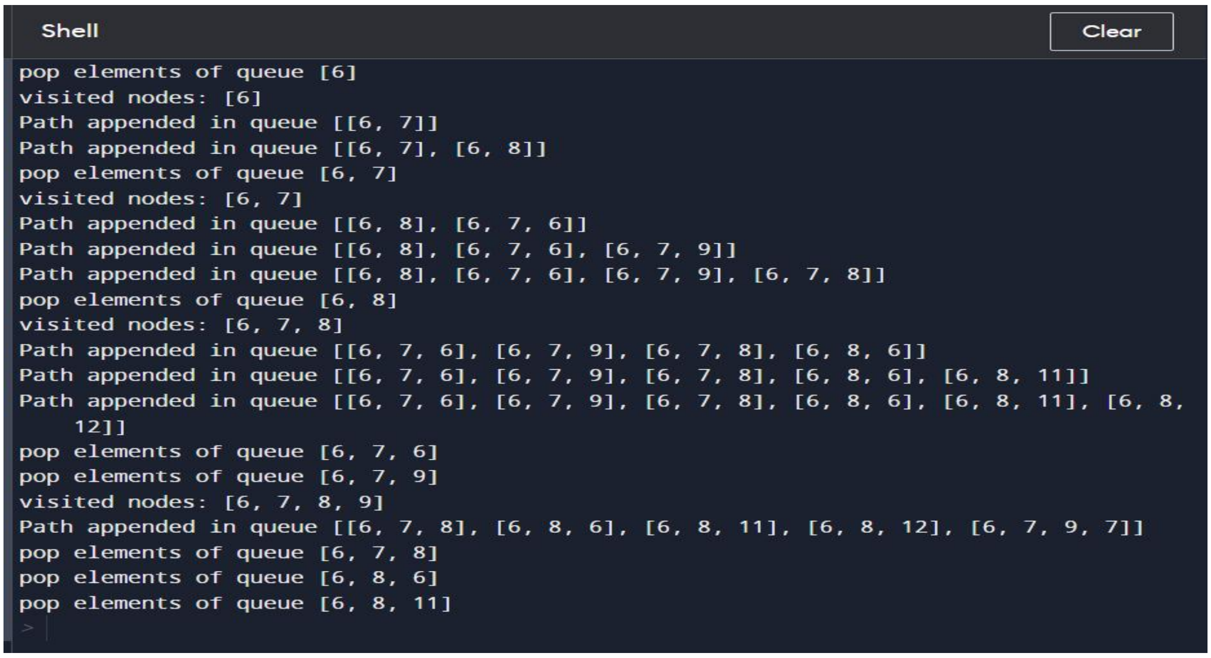
```
a popped
b popped
e popped
c popped
d popped
f popped
g popped
All nodes traversed
BFS Traversal: ['a', 'b', 'e', 'c', 'd', 'f', 'g']
>
```

BFS Goal:

SOURCE CODE:

```
def bfs(graph, start, end):
    visited=[]
    queue =[[start]]
    if start==end:
        print('start and end are same')
    while queue:
        pathvertices=queue.pop(0)
        print("pop elements of queue", pathvertices)
        vertex=pathvertices[-1]
        if vertex not in visited:
            visited.append(vertex)
            print("visited nodes:", visited)
            neighbours=graph[vertex]
            for n in neighbours:
                newpath=list(pathvertices)
                newpath.append(n)
                queue.append(newpath)
                print("Path appended in queue",queue,"\n")
                if n==end:
                    return(newpath)

graph1={6:[7,8],7:[6,9,8],8:[6,11,12],9:[7],10:[7],11:[8],12:[8]}
v3 = bfs(graph1, 6, 10)
print('Shortest path is:', v3)
```

OUTPUT:



```
pop elements of queue [6]
visited nodes: [6]
Path appended in queue [[6, 7]]
Path appended in queue [[6, 7], [6, 8]]
pop elements of queue [6, 7]
visited nodes: [6, 7]
Path appended in queue [[6, 8], [6, 7, 6]]
Path appended in queue [[6, 8], [6, 7, 6], [6, 7, 9]]
Path appended in queue [[6, 8], [6, 7, 6], [6, 7, 9], [6, 7, 8]]
pop elements of queue [6, 8]
visited nodes: [6, 7, 8]
Path appended in queue [[6, 7, 6], [6, 7, 9], [6, 7, 8], [6, 8, 6]]
Path appended in queue [[6, 7, 6], [6, 7, 9], [6, 7, 8], [6, 8, 6], [6, 8, 11]]
Path appended in queue [[6, 7, 6], [6, 7, 9], [6, 7, 8], [6, 8, 6], [6, 8, 11], [6, 8,
    12]]
pop elements of queue [6, 7, 6]
pop elements of queue [6, 7, 9]
visited nodes: [6, 7, 8, 9]
Path appended in queue [[6, 7, 8], [6, 8, 6], [6, 8, 11], [6, 8, 12], [6, 7, 9, 7]]
pop elements of queue [6, 7, 8]
pop elements of queue [6, 8, 6]
pop elements of queue [6, 8, 11]
>
```

PRACTICAL NO: 2

AIM:- Implementing un-informed search strategies. Depth First Search (DFS) and Goal based DFS

THEORY:- Depth First Search or DFS for a Graph

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

- Uses a stack to keep track of the nodes to visit.
- Recursive or iterative implementations are possible.
- May not find the shortest path.

Goal-Based Depth First Search:

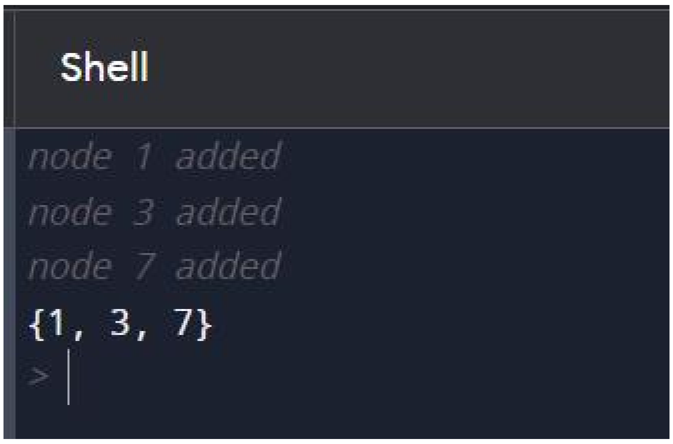
Goal-Based DFS is a modification of the traditional DFS algorithm that is focused on finding a particular goal node or state in a search space.

- Similar to DFS, but with a goal condition to stop the search.
- Useful in problem-solving scenarios where the objective is to find a specific solution.
- May not always find the optimal solution.

SOURCE CODE:

```
def dfs(graph,start,goal):
    visited = set()
    stack = [start]
    path=[]
    while stack:
        node = stack.pop()
        if node not in path:
            path.append(node)
        if node not in visited:
            visited.add(node)
            print('node',node,"added")
        if node ==goal:
            return visited
        for neighbor in graph[node]:
            if neighbor not in visited:
                stack.append(neighbor)
graph1 = {1:[2,3],2:[1,4,5],3:[1,6,7],4:[2],5:[2],6:[3],7:[3]}
visited = dfs(graph1,1,7)
print(visited)
```

OUTPUT:-

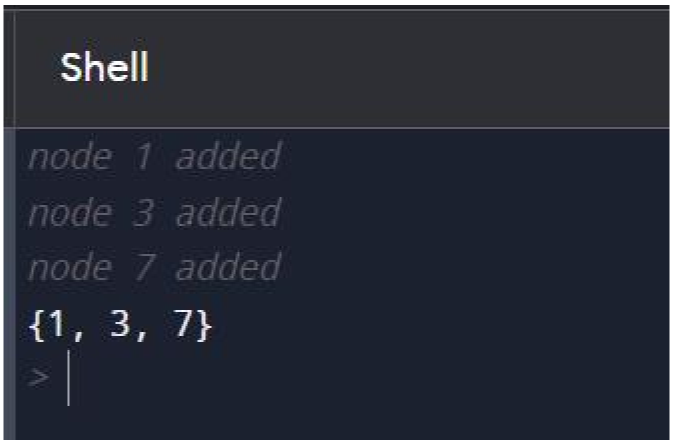


```
node 1 added
node 3 added
node 7 added
{1, 3, 7}
>
```

DFS Goal:

```
def dfs(graph,start,goal):
    visited = set()
    stack = [start]
    path=[]
    while stack:
        node = stack.pop()
        if node not in path:
            path.append(node)
        if node not in visited:
            visited.add(node)
            print('node',node,"added")
        if node ==goal:
            return visited
        for neighbor in graph[node]:
            if neighbor not in visited:
                stack.append(neighbor)
graph1 = {1:[2,3],2:[1,4,5],3:[1,6,7],4:[2],5:[2],6:[3],7:[3]}
visited = dfs(graph1,1,7)
print(visited)
```

OUTPUT:-



```
node 1 added
node 3 added
node 7 added
{1, 3, 7}
>
```

PRACTICAL NO:3

AIM:- Implementing un-informed search strategies.(Uniform cost search)

THEORY:- Uniform-Cost Search

Definition:

Uniform Cost Search is an algorithm used for traversing or searching a weighted tree or graph. Unlike breadth-first search or depth-first search, UCS evaluates nodes based on the total cost of reaching them from the start node. It always selects the node with the lowest cost to explore next.

Algorithm:

- Initialize with the start node.
- Expand the node with the lowest cost.
- Update the cost of reaching neighboring nodes.
- Repeat until the goal node is reached or the search space is exhausted.

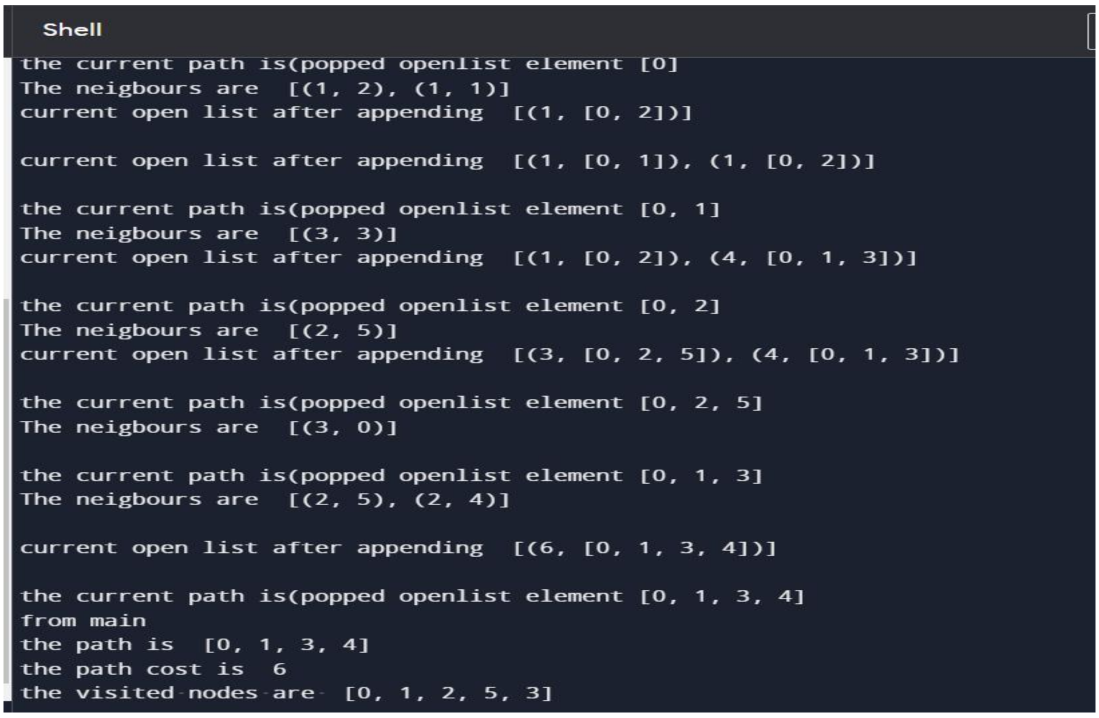
Key Points:

- Utilizes a priority queue to always select the lowest-cost node.
- Guarantees finding the least-cost path to a goal node.
- Requires non-negative edge costs.
- More informed than simple breadth-first or depth-first search, as it considers the cost of reaching each node.

SOURCE CODE:

```
import copy
def uniform_cost_search(graph,start,goal):
    path=[]
    visited=[start]
    path_cost=0
    if start == goal:
        return path,path_cost,visited
    path.append(start)

    openlist=[(path_cost,path)]
    while len(openlist)>0:
        currcost,currpath=openlist.pop(0)
        print('the current path is(popped openlist element',currpath)
        currnode=currpath[-1]
        if currnode ==  goal:
            return currpath,currcost,visited

        if currnode not in visited:
            visited.append(currnode)

        neighbours=graph[currnode]
        print('The neigbours are ',neighbours)
        for n in neighbours:
            n_path_cost=currcost+n[0]
            n_path=copy.copy(currpath)
            n_path.append(n[1])
            n_openlist_ele=(n_path_cost,n_path)
            if n[1] not in visited:
                openlist.append(n_openlist_ele)
                openlist.sort()
                print("current open list after appending ",openlist)
            print("")
    return path,n_path_cost,visitd


graph={0:[(1,2),(1,1)],2:[(2,5)],1:[(3,3)],3:[(2,5),(2,4)],4:[(1,5)],5:[(3,0)]}
p,c,v=uniform_cost_search(graph,0,4)
print("from main")
print("the path is ",p)
print("the path cost is ",c)
print("the visited nodes are ",v)
```

OUTPUT:-



```
the current path is(popped openlist element [0]
The neigbours are  [(1, 2), (1, 1)]
current open list after appending  [(1, [0, 2])]

current open list after appending  [(1, [0, 1]), (1, [0, 2])]

the current path is(popped openlist element [0, 1]
The neigbours are  [(3, 3)]
current open list after appending  [(1, [0, 2]), (4, [0, 1, 3])]

the current path is(popped openlist element [0, 2]
The neigbours are  [(2, 5)]
current open list after appending  [(3, [0, 2, 5]), (4, [0, 1, 3])]

the current path is(popped openlist element [0, 2, 5]
The neigbours are  [(3, 0)]

the current path is(popped openlist element [0, 1, 3]
The neigbours are  [(2, 5), (2, 4)]

current open list after appending  [(6, [0, 1, 3, 4])]

the current path is(popped openlist element [0, 1, 3, 4]
from main
the path is  [0, 1, 3, 4]
the path cost is  6
the visited nodes are  [0, 1, 2, 5, 3]
```

<u>PRACTICAL NO: 4</u>

AIM:- Implementing informed (Heuristic) search strategies. A* search

THEORY:- A* Search Algorithm

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.
What A* Search Algorithm does is that at each step it picks the node according to a value-'**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and process that node/cell.
We define '**g**' and '**h**' as simply as possible below
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final

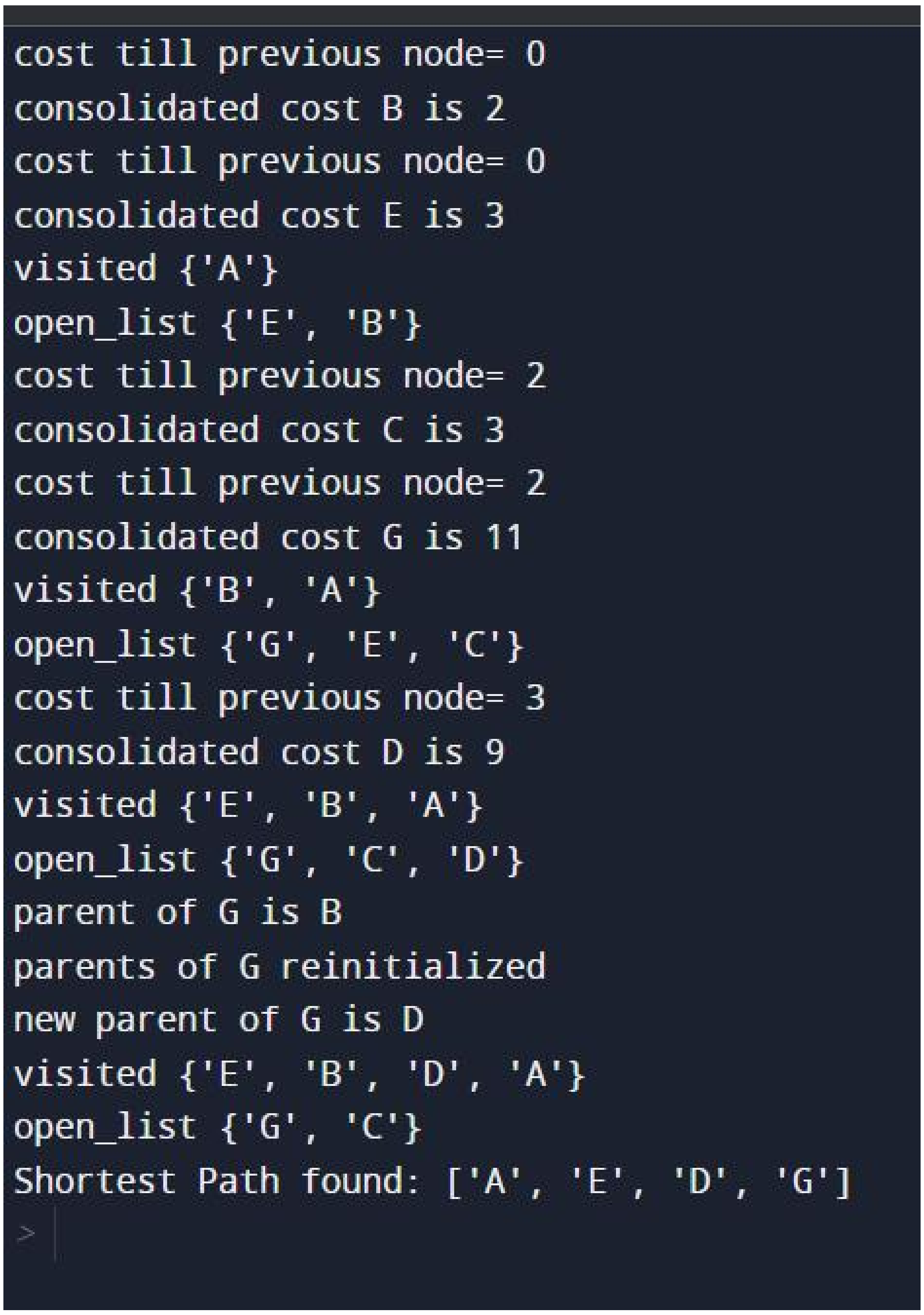
destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.).

SOURCE CODE:-

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}

    g[start_node] = 0
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None

        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):

                n = v

        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m,weight) in get_neighbors(n):

                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                    print('cost till previous node=',g[n])
                    print('consolidated cost',m,'is',g[m])

                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight

                    print('parent of',m,'is',parents[m])
                    parents[m] =n
                    print('parents of',m,'reinitialized')
                    print('new parent of',m,'is',parents[m])

                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None
        if n == stop_node:
            path = []

            while parents[n] != n:
                path.append(n)
                n = parents[n]

            path.append(start_node)

            path.reverse()

            print('Shortest Path found: {}'.format(path))
            return path

        open_set.remove(n)
        closed_set.add(n)
        print('visited',closed_set)
        print('open_list',open_set)

    print('Path does not exist!')
    return None

def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
        'D': 1,
        'E': 7,
        'G': 0,

    }

    return H_dist[n]

Graph_nodes = {
    'A': [('B',2), ('E',3)],
    'B': [('C',1), ('G',9)],
    'C': None,
    'E': [('D',6)],
    'D': [('G',1)],

}
```

aStarAlgo('A','G')

OUTPUT:-



```
cost till previous node= 0
consolidated cost B is 2
cost till previous node= 0
consolidated cost E is 3
visited {'A'}
open_list {'E', 'B'}
cost till previous node= 2
consolidated cost C is 3
cost till previous node= 2
consolidated cost G is 11
visited {'B', 'A'}
open_list {'G', 'E', 'C'}
cost till previous node= 3
consolidated cost D is 9
visited {'E', 'B', 'A'}
open_list {'G', 'C', 'D'}
parent of G is B
parents of G reinitialized
new parent of G is D
visited {'E', 'B', 'D', 'A'}
open_list {'G', 'C'}
Shortest Path found: ['A', 'E', 'D', 'G']
>
```

PRACTICAL NO: 5

AIM:- Implementing backtracking search for constraint satisfaction problem.(CSP).

THEORY:- Constraint Satisfaction Problems (CSP) in Artificial Intelligence

Finding a solution that meets a set of constraints is the goal of constraint satisfaction problems (CSPs), a type of AI issue. Finding values for a group of variables that fulfill a set of restrictions or rules is the aim of constraint satisfaction problems. For tasks including resource allocation, planning, scheduling, and decision-making, CSPs are frequently employed in AI.

There are mainly three basic components in the constraint satisfaction problem:
Variables: The things that need to be determined are variables. Variables in a CSP are the objects that must have values assigned to them in order to satisfy a particular set of constraints. Boolean, integer, and categorical variables are just a few examples of the various types of variables variables, for instance, could stand in for the many puzzle cells that need to be filled with numbers in a sudoku puzzle.

Domains: The range of potential values that a variable can have is represented by domains. Depending on the issue, a domain may be finite or limitless. For instance, in Sudoku, the set of numbers from 1 to 9 can serve as the domain of a variable representing a problem cell.

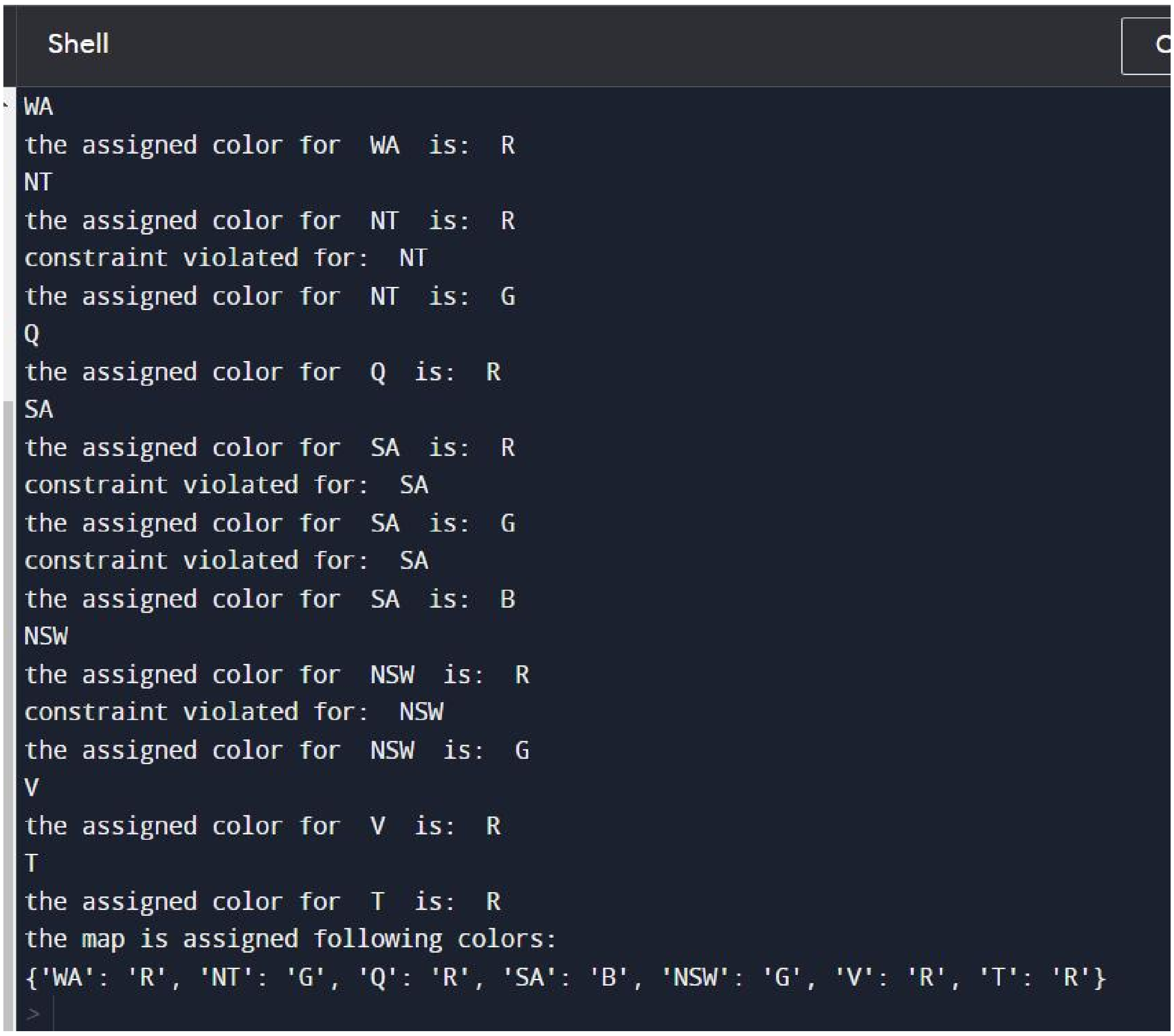
Constraints: The guidelines that control how variables relate to one another are known as constraints. Constraints in a CSP define the ranges of possible values for variables. Unary constraints, binary constraints, and higher-order constraints are only a few examples of the various sorts of constraints. For instance, in a sudoku problem, the restrictions might be that each row, column, and 3×3 box can only have one instance of each number from 1 to 9.

SOURCE CODE:-

```
domain=['R','G','B']

assigned={'WA':None,'NT':None,'Q':None,'SA':None,'NSW':None,'V':None,'T':None}

neighbours={'WA':['NT','SA'],'NT':['WA','Q','SA'],'Q':['NT','SA','NSW'],

        'SA':['WA','NT','Q','NSW','V'],'NSW':['Q','SA','V'],

        'V':['SA','NSW'],'T':[]}

def assignCol():

    for key in assigned.keys():

        print(key)

        i,flag=0,0
        current=key
        if assigned[key]==None:
            while i<=(len(domain)-1):
                assigned[current]=domain[i]


                print('the assigned color for ',current,' is: ',assigned[current])
                flag=checkConst(current)
                if flag==1:
                    i=i+1
                else:
                    break
def checkNbr(key):
    for k,v in neighbours.items():
        if k==key:
            return neighbours[key]


def checkConst(key):
    flag=0
    nblist=checkNbr(key)
    for value in nblist:
        if assigned[key]==assigned[value]:
            print('constraint violated for: ',key)
            flag=1
            return flag
```

```
assignCol()

print('the map is assigned following colors: ')
print(assigned)
```

OUTPUT:-



```
WA
the assigned color for  WA  is:  R
NT
the assigned color for  NT  is:  R
constraint violated for:  NT
the assigned color for  NT  is:  G
Q
the assigned color for  Q  is:  R
SA
the assigned color for  SA  is:  R
constraint violated for:  SA
the assigned color for  SA  is:  G
constraint violated for:  SA
the assigned color for  SA  is:  B
NSW
the assigned color for  NSW  is:  R
constraint violated for:  NSW
the assigned color for  NSW  is:  G
V
the assigned color for  V  is:  R
T
the assigned color for  T  is:  R
the map is assigned following colors:
{'WA': 'R', 'NT': 'G', 'Q': 'R', 'SA': 'B', 'NSW': 'G', 'V': 'R', 'T': 'R'}
>
```