K. J. Somaiya College of Engineering, Mumbai-77

(A Constituent College of Somaiya Vidyavihar University)

Department of Electronics Engineering

Course Name: **Modern Artificial Intelligence**

Semester: **IV**

Experiment No: **TWO**

Title: **Implement Breadth First Search for a search tree**

Date of Performance: **29/01/2024**

Faculty Name:
Prof. Umang Patel

Name and Roll No.:

Vrishank Warrier, 16014022096

Contents

Experiment: TWO
Implement Breadth First Search for a search tree

Vrishank Warrier, 16014022096

29/01/2024

1 Aim and Objective of the Experiment

- Objective 1: For a given goal test, display the path from the start state to the goal state.
- Objective 2: Calculate time and space complexity.
- Objective 3: Implement BFS for real-world problems:
  1. Vacuum cleaner robot
  2. 8-puzzle game
  3. Map of Romania

2 Theory

Breadth First Search (BFS) is like searching a tree by exploring all its branches level by level, starting from the top. It checks all the immediate neighbors before moving to the next level. Using a "first in, first out" approach (like waiting in line), BFS visits nodes in the order they were discovered, making it useful for finding the shortest path in certain scenarios and systematically exploring graphs or trees.

3 Code

3.1 Tool Used/Language

1. VS CODE
2. PYTHON
3. OVERLEAF

3.2 Programs

```
counter = 0

class tree:
    def __init__(self, value):
        self.val = value
        self.children = []

def bfs(root, search_value):
    global counter
    if root is None:
        return None

    queue = [root]

    while queue:
        counter += 1
        node = queue.pop(0)
        if search_value == node.val:
            return node.val

        for child in node.children:
            queue.append(child)

root = tree(1)
root.children = [tree(2), tree(3), tree(4)]
root.children[0].children = [tree(5), tree(6)]
root.children[2].children = [tree(7), tree(8)]

print("Implementing BFS for search tree")
search_value = int(input("Enter value you want to search for: "))

x = bfs(root, search_value)
print("Node found:", x)
print("Time Complexity:", counter)
```

Listing 1: Implementing BFS for search tree

2.
```python
from collections import deque

def get_neighbors(state):
    neighbors = []
    zero_index = state.index(0)
    if zero_index >= 3:
        new_state = state.copy()
        new_state[zero_index], new_state[zero_index - 3] = new_state[zero_index - 3], new_state[zero_index]
        neighbors.append(new_state)
    if zero_index < 6:
        new_state = state.copy()
        new_state[zero_index], new_state[zero_index + 3] = new_state[zero_index + 3], new_state[zero_index]
        neighbors.append(new_state)
    if zero_index % 3 != 0:
        new_state = state.copy()
        new_state[zero_index], new_state[zero_index - 1] = new_state[zero_index - 1], new_state[zero_index]
        neighbors.append(new_state)
    if (zero_index + 1) % 3 != 0:
        new_state = state.copy()
        new_state[zero_index], new_state[zero_index + 1] = new_state[zero_index + 1], new_state[zero_index]
        neighbors.append(new_state)
    return neighbors

def bfs(initial_state, final_state):
    visited = set()
    queue = deque([(initial_state, [])])

    while queue:
        state, path = queue.popleft()
        visited.add(tuple(state))

        if state == final_state:
            return path

        for neighbor in get_neighbors(state):
            if tuple(neighbor) not in visited:
                queue.append((neighbor, path + [neighbor]))

    return None


def main():
    print("Enter initial state of the puzzle (0 represents the empty tile):")
    initial_state = [int(input()) for _ in range(9)]
```

```
    print("Enter final state of the puzzle:")
    final_state = [int(input()) for _ in range(9)]

    path = bfs(initial_state, final_state)

    if path:
        print("Shortest path to reach the final state:")
        step = 1
        for state in path:
            print("Step", step)
            print(state[:3])
            print(state[3:6])
            print(state[6:])
            print()
            step += 1
    else:
        print("No solution exists for the given states.")


main()
```

Listing 2: Implementing BFS for 8 puzzle game

3.3 Output

1. Output of bfs

```
PS C:\Users\91993> & C:/Users/91993/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/91993/Desktop/MAI/EXPERIMENTS/EXP2/BFS.py
Implementing BFS for search tree
Enter value you wanna search for: 4
Node found: 4
PS C:\Users\91993> & C:/Users/91993/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/91993/Desktop/MAI/EXPERIMENTS/EXP2/BFS.py
Implementing BFS for search tree
Enter value you wanna search for: 8
Node found: 8
PS C:\Users\91993> & C:/Users/91993/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/91993/Desktop/MAI/EXPERIMENTS/EXP2/BFS.py
Implementing BFS for search tree
Enter value you wanna search for: 99
Node found: None
PS C:\Users\91993>
```

2. Output of 8 puzzle game

```
Enter initial state of the puzzle (0 represents the empty tile):
2
8
3
1
0
4
7
6
5
Enter final state of the puzzle:
1
2
3
8
0
4
7
6
5
Shortest path to reach the final state:
Step 1
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Step 2
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Step 3
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Step 4
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

PS C:\Users\91993\Desktop\MAI>
```

3.4 Time and Space Complexity

- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$

4 Post Lab Question

1. Explain why BFS is worst approach when the branching factor and solution depth in state-space tree is large (value =10 or more).
   ANS: BFS can use a lot of memory because it stores many nodes in a queue, especially when there are lots of choices and the solution is deep. For instance, if there are 10 options at each step and the solution is 10 steps away, BFS might need to store millions of nodes.

   Additionally, BFS can be very slow when both the number of choices and the solution depth are high. For example, with 10 choices at each step and a solution 10 steps deep, BFS might need to explore billions of nodes, making it very slow.

2. True or False? Justify - BFS is complete even if zero step costs are allowed.
   ANS: True.
   BFS is complete even when zero step costs are allowed because it exhaustively explores all nodes at each level, ensuring it will eventually find the shallowest goal state if one exists.

3. Prove or give counterexamples. - BFS is a special case of uniform-cost search.
   ANS: Breadth-first search (BFS) can be considered a special case of uniform-cost search (UCS) when all step costs are uniform, i.e., every step from one node to another has the same cost.

   In BFS, the cost of each step is uniform, as it doesn't consider the cost of reaching each node. It explores nodes level by level, ensuring that it finds the shallowest goal state. This is equivalent to UCS with a constant step cost, where the cost of reaching each node is the same.

   Therefore, BFS is indeed a special case of uniform-cost search when all step costs are uniform.

5 Conclusion

We have successfully learned about BFS Search algorithms and implemented the algorithm and were able to understand its working. Using BFS, we were also able to implement a real-life application, i.e. Solving 8 puzzle game.

References

[1] Breadth first search. https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/.

[2] Breadth first search or bfs for a graph. https://medium.com/@sergioli/breath-first-and-depth-first-search-on-tree-and-graph-in-python-99fd1861893e#:~:text=Breath%2Dfirst%20Search,height%20'h%2B1'/.