1. How is BFS/DFS with extended list different from normal DFS/BFS?

**Normal BFS/DFS**:

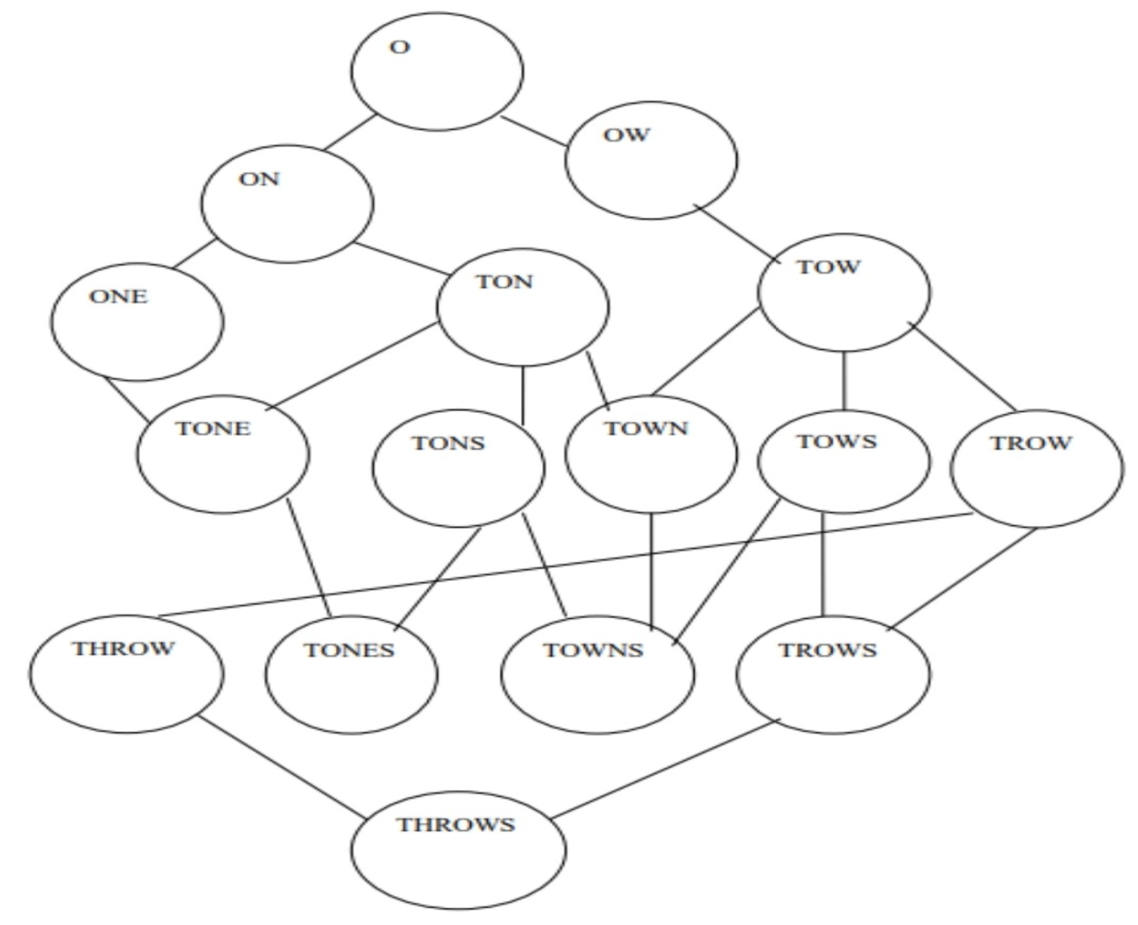
* + **BFS**: It explores all the nodes at the present depth level before moving on to the nodes at the next depth level. It uses a queue to keep track of the next location to visit.
  + **DFS**: It explores as far as possible along each branch before backtracking. It typically uses a stack or recursion for implementation.

**BFS/DFS with Extended List**:

* + **Extended List**: This is an additional data structure used to keep track of nodes that have already been visited. It is often a set or a list.
  + **Purpose**: The primary purpose of the extended list is to avoid revisiting nodes, which can be crucial in large or infinite graphs to prevent infinite loops and to reduce time complexity.
  + **Operation**: When a node is first visited, it is added to the extended list. Before visiting a new node, the algorithm checks if the node is in the extended list. If it is, the node is skipped; if not, the node is processed and added to the extended list.
  + **Effect on BFS/DFS**:
    - **BFS with Extended List**: It ensures that each node is visited only once, which is particularly beneficial when there are many redundant paths to the same node.
    - **DFS with Extended List**: Similarly, it prevents revisiting nodes, making DFS more efficient in scenarios with loops or redundant paths.

In summary, the use of an extended list in BFS/DFS is a practical enhancement to manage and optimize the search, especially in complex graphs where nodes might be revisited through different paths.

Q2) You and some friends are playing a game where the object is to build up a 6-letter word starting with a single letter and adding or subtracting one letter each turn. To make the game more interesting, each intermediate step along the way must also be a word, and all the words are limited to those in a specified list (O, ON, OW, ONE, THROW, TON, TONE, TONES, TONS, TOW, TOWN, TOWNS, TOWS, TROW, TROWS, and THROWS). This leads to a search problem as mentioned in the graph below:



**Answer-2.** Rajul’s approach using Breadth-First Search with extended list would find the following path: O -> ON -> ONE -> TONE -> THROW

**Answer-3.** Rutul’s approach using Depth-First Search with backtracking and lexicographic order would require him to backtrack 4 times. His final path would be: O -> OW -> TOWN -> THROW.

Depth-First Search (DFS) explores one branch as far as possible before backtracking and trying another branch.

-> With backtracking and lexicographic order, Rutul would prioritize exploring words that come alphabetically before others at the same level.

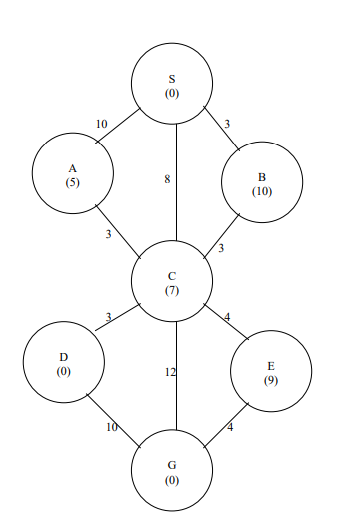
-> He would first explore O -> OW -> TONE, but then have to backtrack because TONE has already been visited.

-> Next, he would explore O -> OW -> TOWN, and then backtrack again because TOWN has already been visited.

-> He would then explore O -> OW -> TOWS, but backtrack once more because TOWS has already been visited.

-> Finally, he would explore O -> OW -> TOWN -> THROW, which is a valid path and lexicographically the first one he found.

Apply the A\* algorithm to find the path from S to G. Heuristic values in each node are mentioned in parenthesis. Mention what path will be found with justification.



Path: S → B → C → G

Justification:

1. Initial State:

- A\* starts at S with g(S) = 0 (no cost to reach S itself) and f(S) = 0 + h(S) = 0 (estimated total cost).

2. Expanding Nodes:

- A\* considers the neighbors of S: A and B.

- For A: f(A) = g(S) + 10 + h(A) = 10 + 5 = 15.

- For B: f(B) = g(S) + 3 + h(B) = 3 + 10 = 13.

- A\* prioritizes B because it has a lower f-value (13 < 15).

3. Exploring B:

- A\* updates g(B) = 3 (actual cost to reach B) and f(B) = 3 + h(B) = 13.

- A\* considers the neighbors of B: C and E.

- For C: f(C) = g(B) + 3 + h(C) = 6 + 7 = 13.

- For E: f(E) = g(B) + 4 + h(E) = 7 + 9 = 16.

- A\* prioritizes C because it has a lower f-value (13 < 16).

4. Exploring C:

- A\* updates g(C) = 6 (actual cost to reach C) and f(C) = 6 + h(C) = 13.

- A\* finds the goal node G as a neighbor of C.

- f(G) = g(C) + 10 + h(G) = 16 + 0 = 16.

- Despite having a higher f-value, G is selected as the goal has been reached.

Therefore, the A\* algorithm finds the path S → B → C → G with a total cost of 16.

Key Points:

- A\* prioritizes nodes with lower f-values, which balance actual path cost (g) and estimated remaining cost (h).

- The heuristic values guided the search towards the goal, as nodes closer to G generally had lower h-values.

- A\* found the optimal path in this case, as the heuristic was admissible (never overestimated actual costs).