

Topic Introduction

Today's theme: **BFS for Shortest Paths in Word and Mutation Problems**

One of the most powerful tools in your coding interview toolkit is the **Breadth-First Search (BFS)** algorithm. BFS is a graph traversal technique that explores all nodes (or possibilities) at the current "depth" before moving on to nodes at the next depth level. In other words, it checks all immediate neighbors before progressing further.

What exactly is BFS?

BFS starts from a "source" node and explores all the nodes that are one step away, then all nodes that are two steps away, and so on. It uses a **queue** to keep track of what to visit next. This property makes BFS ideal for finding shortest paths in unweighted graphs, because the first time you reach a node, you've done so via the shortest possible path.

When do you use BFS?

Think about problems where you have to find the minimum number of steps or the shortest transformation from a start state to an end state, given a set of allowed moves. BFS is both intuitive and optimal for such cases.

Simple Example (Not one of our target problems!):

Imagine a chessboard. If you want to find the minimum number of moves for a knight to reach a specific square, you'd use BFS. Each board position is a node, and each legal knight move is an edge.

Now, let's look at three classic problems that all use BFS for shortest paths, but with a twist: instead of moving on a board, you're transforming words or genetic codes by changing one letter at a time.

Why These Three Problems?

- **Word Ladder** asks you to find the shortest transformation sequence length from a start word to an end word, changing one letter at a time, and each intermediate word must be in a dictionary.
- **Word Ladder II** wants *all* the shortest transformation sequences, not just the length.
- **Minimum Genetic Mutation** is essentially Word Ladder with genetic codes and a mutation bank.

They all require building a graph where words (or sequences) are nodes, and edges exist if you can transform one to another in a single change. The challenge is finding the shortest path from start to end, sometimes returning just the length, sometimes all paths.

Let's dig into each one.

Problem 1: Minimum Genetic Mutation

[LeetCode 433. Minimum Genetic Mutation](#)

Problem Statement (in my words):

Given a start genetic string, an end genetic string, and a list of valid gene mutations (the bank), find the minimum number of mutations needed to change the start into the end. Each mutation must change exactly one character, and each intermediate string must be in the bank. If it's impossible, return -1.

PrepLetter: Word Ladder and similar

Example:

Input:

start = "AACCGGTT"

end = "AACCGGTA"

bank = ["AACCGGTA"]

Output: 1

Explanation: You can change the last letter from 'T' to 'A'.

Another Test Case for You:

start = "AACCGGTT"

end = "AAACGGTA"

bank = ["AACCGGTA", "AACCGCTA", "AAACGGTA"]

(What's the answer? Try it with pen and paper!)

Brute Force Approach:

Try every possible sequence of mutations, recursively, to see if you can reach the end. This is exponential in the worst case since you try all combinations!

Time: $O(2^N)$, too slow.

Optimal Approach: BFS

- Each gene string is a node.
- There's an edge between two nodes if you can mutate one into the other by changing a single letter and the result is in the bank.
- BFS guarantees the first time you reach the end, it is via the shortest path.

Step-by-Step:

- Put the start string in a queue, with a step counter (depth).
- At each step, for the current string, generate all possible valid mutations (by changing each character to one of 'A', 'C', 'G', 'T').
- If the mutation is in the bank and not visited, add it to the queue.
- Mark visited mutations to avoid cycles.
- If you reach the end string, return the number of steps.
- If the queue is empty and you haven't found the end, return -1.

Python Solution:

```
from collections import deque

def minMutation(start, end, bank):
    """
    Returns the minimum number of mutations to transform start into end.
    Each mutation changes a single character, and must be in bank.
    """
    bank_set = set(bank)
    if end not in bank_set:
        return -1 # end must be in the bank to be reachable

    # Valid gene characters
```

```
genes = ['A', 'C', 'G', 'T']
queue = deque()
queue.append((start, 0)) # (current_string, mutation_count)
visited = set([start])

while queue:
    current, steps = queue.popleft()
    if current == end:
        return steps

    # Try all one-letter mutations
    for i in range(len(current)):
        for g in genes:
            if current[i] == g:
                continue
            mutated = current[:i] + g + current[i+1:]
            if mutated in bank_set and mutated not in visited:
                visited.add(mutated)
                queue.append((mutated, steps + 1))
return -1 # Not found
```

Time Complexity:

- $O(N * M)$, where N is the length of the bank and M is the length of each gene string (since for each string, we try changing each character).

Space Complexity:

- $O(N)$, for the queue and the visited set.

How does this code work?

- It uses a queue to explore strings level by level.
- For each string, it tries all one-character mutations.
- If the mutation is valid (in bank and not visited), it's queued for the next BFS layer.
- The process ends successfully when the end string is dequeued, otherwise returns -1.

Trace of Example:

start = "AACCGGTT", end = "AACCGGTA", bank = ["AACCGGTA"]

- Start: "AACCGGTT" (step 0)
 - Try changing each letter. Only "AACCGGTA" (change last T to A) is in the bank.
 - Queue: ("AACCGGTA", 1)
- Dequeue: "AACCGGTA" (step 1)
 - It's the end! Return 1.

Try This Test Case:

start = "AAAAACCC", end = "AACCCCCC", bank = ["AAAACCCC", "AAACCCCC", "AACCCCCC"]

Take a moment to solve this on your own before jumping into the solution.

Problem 2: Word Ladder

[LeetCode 127. Word Ladder](#)

How is this similar to Problem 1?

It's the same pattern: change one character at a time, each intermediate word must be in the dictionary (word list), and find the shortest transformation sequence from beginWord to endWord.

Problem Statement (rephrased):

Given a beginWord, endWord, and a wordList, find the minimum number of single-letter transformations from beginWord to endWord, where each intermediate word must be in wordList. Each transformation can only change one letter at a time.

Example:

Input:

beginWord = "hit"

endWord = "cog"

wordList = ["hot", "dot", "dog", "lot", "log", "cog"]

Output: 5

Explanation: "hit" -> "hot" -> "dot" -> "dog" -> "cog"

Another Test Case for You:

beginWord = "hit"

endWord = "cog"

wordList = ["hot", "dot", "dog", "lot", "log"]

(What should the answer be?)

Brute Force Approach:

Try all possible transformation paths. This is exponential and impractical for even moderate word lists.

Optimal Approach: BFS

Exactly the same reasoning as Problem 1:

- Each word is a node.
- There's an edge if you can transform one word to another in one letter and the new word is in the word list.
- BFS ensures you find the shortest path.

Step-by-Step:

- Use a queue for BFS, starting with (beginWord, steps=1).
- For each word at the front of the queue:
 - Try changing each character to every letter from 'a' to 'z'.
 - If the new word is in the word list and not visited, add it to the queue.
 - Mark words as visited.
- When you reach endWord, return the steps.
- If the queue is empty and endWord not found, return 0.

Pseudocode:

```
function ladderLength(beginWord, endWord, wordList):  
    wordSet = set(wordList)
```

```
if endWord not in wordSet:  
    return 0  
  
queue = [(beginWord, 1)]  
visited = set([beginWord])  
  
while queue is not empty:  
    currentWord, steps = queue.pop_front()  
    if currentWord == endWord:  
        return steps  
    for each position i in currentWord:  
        for each letter c from 'a' to 'z':  
            if c == currentWord[i]:  
                continue  
            mutatedWord = currentWord with position i replaced by c  
            if mutatedWord in wordSet and mutatedWord not in visited:  
                visited.add(mutatedWord)  
                queue.append((mutatedWord, steps + 1))  
return 0
```

Example Trace:

beginWord = "hit", endWord = "cog", wordList = ["hot", "dot", "dog", "lot", "log", "cog"]

- "hit" -> "hot" (step 2)
- "hot" -> "dot" -> "lot"
- "dot" -> "dog" (step 4)
- "dog" -> "cog" (step 5)

So, "hit" -> "hot" -> "dot" -> "dog" -> "cog" (5 steps)

Try This Test Case:

beginWord = "a", endWord = "c", wordList = ["a", "b", "c"]

Complexity:

- Time: $O(N * M^2)$ where N is the number of words and M is the length of each word (since for each word, we try all possible letter substitutions).
- Space: $O(N)$ for queue and visited set.

Problem 3: Word Ladder II

[LeetCode 126. Word Ladder II](#)

What's different?

You don't just want the length of the shortest path—you want *all possible shortest transformation sequences*. This means you need to track entire paths, not just the current word and step count.

PrepLetter: Word Ladder and similar

Problem Statement (in my words):

Given a beginWord, an endWord, and a wordList, return all shortest transformation sequences from beginWord to endWord, changing one letter at a time and only using words from the wordList. Each sequence must be as short as possible.

Example:

Input:

beginWord = "hit"

endWord = "cog"

wordList = ["hot", "dot", "dog", "lot", "log", "cog"]

Output:

```
[  
    ["hit","hot","dot","dog","cog"],  
    ["hit","hot","lot","log","cog"]  
]
```

Another Test Case for You:

beginWord = "red"

endWord = "tax"

wordList = ["ted", "tex", "red", "tax", "tad", "den", "rex", "pee"]

Brute Force Approach:

Try all possible paths—a recipe for timeouts.

Optimal Approach: BFS with Path Tracking

- Use BFS to build a tree of shortest paths.
- For each word, at each BFS level, keep track of all paths that reach it (not just the first).
- Stop when you reach the endWord at the first time (shortest path). Don't process deeper levels.
- Return all collected paths that reach endWord.

Pseudocode:

```
function findLadders(beginWord, endWord, wordList):  
    wordSet = set(wordList)  
    if endWord not in wordSet:  
        return []  
  
    result = []  
    queue = [[beginWord]]  
    visited = set([beginWord])  
    found = False  
  
    while queue and not found:  
        next_level_visited = set()  
        next_queue = []  
        for path in queue:  
            last_word = path[-1]
```

```
for each position i in last_word:  
    for each letter c from 'a' to 'z':  
        if c == last_word[i]:  
            continue  
        new_word = last_word with position i replaced by c  
        if new_word in wordSet and new_word not in visited:  
            new_path = path + [new_word]  
            if new_word == endWord:  
                result.append(new_path)  
                found = True  
            next_queue.append(new_path)  
            next_level_visited.add(new_word)  
        queue = next_queue  
        visited.update(next_level_visited)  
return result
```

Try This Test Case:

```
beginWord = "a", endWord = "c", wordList = ["a", "b", "c"]
```

Complexity Discussion:

- Time: Can be exponential in the worst case (if many shortest paths), but BFS limits search to shortest paths.
- Space: $O(N * K)$ where N is number of paths and K is the length of each.

Reflect:

- What if you tried DFS? Why would it be less efficient for shortest paths?
- Could you optimize further (bidirectional BFS, preprocessing)?

Summary and Next Steps

These three problems all use **BFS to find shortest paths where nodes are words or gene strings** and edges represent single-character transformations. The subtle differences lie in whether you need just the length, a specific path, or all paths.

Key patterns:

- Model the problem as a graph.
- Use BFS to explore all possible transformations level by level.
- Use a queue and visited set to avoid cycles and redundant work.
- For "all shortest paths," track full paths, not just words.

Common mistakes:

- Not marking nodes as visited at the right time (can cause duplicates or miss paths).
- Failing to check if end state is in the allowed list.
- Using DFS when BFS is needed for shortest paths.

Action List:

- Solve all three problems on your own—even the one with code here.
- For extra challenge: Try implementing Word Ladder II with bidirectional BFS.
- Explore other problems where BFS is used for shortest paths (like sliding puzzles, knight moves, etc).
- Dry-run your solutions on new test cases you invent.
- Compare your code style and edge-case handling with others.
- Don’t worry if you stumble—practice builds intuition. Keep going!

Happy coding!