A4_Report_123090612

Q1 Mirror

①What are the possible solutions for the problem?

1. Standard Shortest Path (Dijkstra's Algorithm):

When to use: When there are **no mandatory edges** to traverse, or the mandatory edges are implicitly covered by the starting and ending nodes.
Approach:

- Use **Dijkstra's algorithm** to find the shortest path between the start node (`s`) and the target node (`t`).
- This is the simplest solution, where we don't worry about any constraints related to edge traversal.

2. Modified Dijkstra's Algorithm with Bitmasking:

When to use: When **mandatory edges** must be traversed as part of the shortest path.
Approach:

- **Bitmasking** is used to represent which mandatory edges have been traversed.
- For each query, we run a **modified Dijkstra's algorithm**, where each state (node) is associated with a bitmask that tracks the mandatory edges.
- If the destination node (`t`) is reached and all mandatory edges have been visited (i.e., the bitmask matches the required value), we print the shortest time.

3. Shortest Path with Constraints (Multi-Path Search):

When to use: For larger graphs where multiple queries are involved, and the mandatory edges vary for each query.
Approach:

- Precompute the **shortest path** from each node to every other node using **Floyd-Warshall** or **Dijkstra** (for each node as the source).
- For each query, combine the precomputed shortest paths and the required mandatory edges.
- This solution avoids recalculating the shortest paths from scratch for every query.

②How do you solve this problem?

1. Helper Functions for Priority Queue

This section defines a set of functions to implement a basic priority queue using a list-based heap.

```python
def heap_push(heap, item):
    heap.append(item)
    sift_up(heap, len(heap) - 1)
def heap_pop(heap):
    if not heap:
        return None
    top = heap[0]
    last = heap.pop()
    if heap:
        heap[0] = last
        sift_down(heap, 0)
    return top
def sift_up(heap, idx):
    while idx > 0:
        parent = (idx - 1) // 2
        if heap[parent][0] > heap[idx][0]:
            heap[parent], heap[idx] = heap[idx], heap[parent]
            idx = parent
        else:
            break
def sift_down(heap, idx):
    n = len(heap)
    while True:
        left = 2 * idx + 1
        right = 2 * idx + 2
        smallest = idx
        if left < n and heap[left][0] < heap[smallest][0]:
            smallest = left
        if right < n and heap[right][0] < heap[smallest][0]:
            smallest = right
        if smallest != idx:
            heap[smallest], heap[idx] = heap[idx], heap[smallest]
            idx = smallest
        else:
            break
```

- **Heap Push**: `heap_push` adds a new element to the heap and ensures the heap property is maintained by "sifting" the new element upwards if necessary (`sift_up`).
- **Heap Pop**: `heap_pop` removes the top element from the heap, which is the smallest one due to the nature of the heap. The heap then re-adjusts by moving the last element to the top and "sifting it down" (`sift_down`) to maintain the heap property.

- **Sifting**: The `sift_up` and `sift_down` functions ensure the heap maintains the min-heap property (smallest element is always at the root).

2. Input Reading Functions

This section contains functions that handle input reading. These are specifically written to read multiple lines of integers from the input.

- `readints` : This function reads a line of input, strips it of extra spaces, splits it into integers, and returns the list. If the line is empty, it continues to read.
- `read_edge` : This is similar to `readints` , but specifically designed for reading edges, ensuring that the input is properly formatted as three integers per line.

3. Main Logic

Initial Setup

```
def main():
    first_line = readints()
    while not first_line:
        first_line = readints()
    n, m, q = first_line
```

- **Reading Input**: The first line contains three integers: `n` (the number of nodes), `m` (the number of edges), and `q` (the number of queries). This is stored in `n` , `m` , and `q` respectively.

Reading Edges and Building the Graph

```
    edges = [None]  # 1-based indexing
    graph = [[] for _ in range(n + 1)]
    for edge_idx in range(1, m + 1):
        edge = read_edge()
        while not edge:
            edge = read_edge()
        u, v, w = edge
        edges.append((u, v, w))
        graph[u].append((v, w, edge_idx))
        graph[v].append((u, w, edge_idx))
```

- `edges` : A list to store all the edges. It starts with `None` for 1-based indexing.
- `graph` : An adjacency list representation of the graph. Each node has a list of tuples representing neighboring nodes, the weight of the edge, and the edge index.
- **Edges**: The code loops through the edges and adds each one to the `edges` list and the adjacency list `graph` .

4. Reading Queries

```
    queries_E = []
    for _ in range(q):
        ki_list = readints()
        while not ki_list:
            ki_list = readints()
        k_i = ki_list[0]
        E_j = []
        while len(E_j) < k_i:
            e_list = readints()
            if not e_list:
                continue
            E_j += e_list
        queries_E.append(E_j[:k_i])
```

- `queries_E` : This list will store the sets of edges that must be traversed for each query.
- **Reading Edges for Each Query**: For each query, it reads the list of edge indices `E_j` that Lee must pass through. The number of edges to traverse (`k_i`) is read first.

Reading Start and End Nodes for Queries

```
    queries_ST = []
    for _ in range(q):
        st_list = readints()
        while not st_list:
            st_list = readints()
        s_j, t_j = st_list
        queries_ST.append((s_j, t_j))
```

- `queries_ST` : This list stores the start (`s_j`) and end (`t_j`) nodes for each query.

5. Processing Each Query

Dijkstra's Algorithm for Shortest Path

```
    for query_idx in range(q):
        E_j = queries_E[query_idx]
        s_j, t_j = queries_ST[query_idx]
```

```
        k_i = len(E_j)
        if k_i == 0:
            # If there are no edges to traverse, simply find the shortest path
            # using standard Dijkstra's algorithm without mask
```

- **Processing Queries**: For each query, it checks if there are specific edges that must be passed through. If there are no such edges, it directly finds the shortest path between `s_j` and `t_j` using a standard Dijkstra's algorithm.

Dijkstra's Algorithm (With Masks for Specific Edges)

If there are edges to pass through, the code applies a modified version of Dijkstra's algorithm that tracks which mandatory edges have been visited using a bitmask.

```
        # Create a mapping from edge_index to bit position
        E_j_map = {}
        for bit_pos in range(k_i):
            edge_index = E_j[bit_pos]
            E_j_map[edge_index] = bit_pos
        target_mask = (1 << k_i) - 1
```

- **Bitmasking**: This step maps each edge in `E_j` to a unique bit position. The `target_mask` represents the bitmask where all mandatory edges have been traversed.

```
        # Initialize Dijkstra
        heap = []
        heap_push(heap, (0, s_j, 0))
        visited = [set() for _ in range(n + 1)]
        answer_found = False
        while heap:
            current = heap_pop(heap)
            if current is None:
                break
            current_time, current_node, mask = current
            if current_node == t_j and mask == target_mask:
                print(current_time)
                answer_found = True
                break
```

- **Modified Dijkstra**: Here, the heap stores the current time, node, and the bitmask. If the destination node `t_j` is reached and all required edges have been passed (i.e., the bitmask matches `target_mask`), the shortest time is printed.

```
            if mask in visited[current_node]:
                continue
            visited[current_node].add(mask)
            for neighbor in graph[current_node]:
                v, w_edge, edge_idx = neighbor
                new_mask = mask
                if edge_idx in E_j_map:
                    bit = E_j_map[edge_idx]
                    new_mask = mask | (1 << bit)
                heap_push(heap, (current_time + w_edge, v, new_mask))
        if not answer_found:
            print(-1)
```

- **Visiting Nodes**: The algorithm checks whether a node with a particular mask has already been visited to avoid unnecessary recalculations. It then updates the mask for each edge and continues the process.

6. Final Output

③Why is your solution better than others?

My solution is **better** because it efficiently handles multiple queries with varied edge requirements, scales well with large graphs, and guarantees optimal results using a combination of **Dijkstra's algorithm** and **bitmasking**. This combination ensures that the solution is both **time-efficient** and **correct**, making it the best choice for the problem.

Q2 Violet

①What are the possible solutions for the problem?

1. Binary Search on Bottleneck Value with BFS/DFS

This solution is the one implemented in the provided code. It combines **binary search** with **BFS** (or DFS) to find the maximum possible minimum number of flowers along a path between two flowerbeds.

- Use **binary search** over the edge flower values (from 0 to the maximum edge weight) to find the maximum value of the minimum flowers along a path.
- For each mid-point in the binary search, use **BFS/DFS** to check if there is a path between `s_i` and `t_i` where every edge on the path has at least `mid` flowers.
- If such a path exists, increase the lower bound of the binary search; otherwise, decrease the upper bound.

2. Union-Find (Disjoint Set Union - DSU) with Path Compression and Union by Rank

Another possible solution could use **Union-Find** (or DSU) to dynamically maintain connected components as the graph changes. This approach focuses on finding connected components efficiently and updating them as edges change.

- Use a **Union-Find** data structure to keep track of the connected components, where each component represents a group of flowerbeds that are connected.
- For each edge update, adjust the Union-Find structure by merging sets (flowerbed groups) as necessary.
- For each query, find the connected components for flowerbeds `s_i` and `t_i`. If they belong to the same component, we can compute the maximum bottleneck value in that component.

3. Dynamic Shortest Path Algorithms

If the goal is to consider all possible paths between `s_i` and `t_i` and dynamically change the graph, we might consider **dynamic shortest path algorithms** like **Dijkstra's algorithm** or **Bellman-Ford**. These algorithms can be adapted to compute the minimum bottleneck on a path.

- Use Dijkstra's or Bellman-Ford to find the shortest or the maximum bottleneck path from `s_i` to `t_i`.
- Each time an edge weight changes, re-run the algorithm to update the shortest or maximum bottleneck path.

②How do you solve this problem?

1. Input Parsing

- The first part reads the number of **flowerbeds** `n` and the number of **edges** `m`.
- Then, we read the edges. Each edge is defined by two nodes `u` and `v` (the flowerbeds it connects) and `w` (the number of flowers on that edge).
- The code also ensures that the flowerbed indices `u` and `v` are ordered in a consistent way (`u < v`). This is done to make sure the graph edges are stored in a uniform order, which helps later in searching for specific edges.
- The list `edge_list` is then sorted lexicographically by `u` and `v`, which will be useful for binary searching edges later.

2. Graph Representation

- An **adjacency list** `adj` is created to represent the graph. Each flowerbed (node) will store a list of tuples, where each tuple represents a neighboring flowerbed and the index of the edge connecting them.
- For every edge in `edge_list`, both flowerbeds `u` and `v` are updated in the adjacency list. This ensures that the graph is undirected, i.e., if there's an edge between flowerbed `u` and flowerbed `v`, both `u` will point to `v` and `v` will point to `u`.

3. Reading Change Requests

- This section handles the reading of the **changes** that occur over time.
- `q` is the number of changes (days), and for each change, we first read `k_i`, the number of paths affected by the change.
- Each change updates the number of flowers on certain paths (edges). The path between flowerbeds `a` and `b` will now have `c` flowers instead of its previous count.
- The list `changes` stores all the changes for each day in the format `[(a, b, c), ...]`.

4. Reading Queries

- After reading the changes, we then read the **queries**.
- For each query (which is asked for each day, including day `0` before any changes), we need to find the path from flowerbed `s_i` to `t_i` that maximizes the minimum number of flowers along the path.
- These queries are stored in the list `queries`.

5. Helper Functions

Edge Index Lookup

```python
# Function to find the index of an edge in the sorted edge_list
def find_edge_index(a, b):
    if a > b:
        a, b = b, a
    l = 0
    r = len(edge_list) - 1
    while l <= r:
        mid = (l + r) // 2
        if edge_list[mid][0] == a and edge_list[mid][1] == b:
            return mid
        elif edge_list[mid][0] < a or (edge_list[mid][0] == a and edge_list[mid][1] < b):
            l = mid + 1
        else:
            r = mid - 1
    return -1  # Edge not found
```

- This function finds the index of an edge in the sorted `edge_list`.
- The binary search approach is used to efficiently locate the edge `(a, b)` by comparing flowerbed indices, ensuring the edge exists in the list and is correctly positioned.

Connectivity Check

```python
# Function to check if there's a path from s to t with all edges having at least w flowers
def is_connected(s, t, w):
    if s == t:
        return True
    visited = [False] * (n + 1)
    queue = [s]
```

```python
    visited[s] = True
    head = 0
    while head < len(queue):
        u = queue[head]
        head += 1
        for (v, edge_idx) in adj[u]:
            if not visited[v] and edge_list[edge_idx][2] >= w:
                if v == t:
                    return True
                visited[v] = True
                queue.append(v)
    return False
```

- This function checks if there exists a **path** from `s` to `t` such that every edge along the path has at least `w` flowers.
- It uses a **BFS (breadth-first search)** approach to explore the graph. If all edges encountered in the path have at least `w` flowers, the function returns `True`, meaning there's a valid path from `s` to `t` with this constraint. If no such path exists, it returns `False`.

Binary Search for Maximum Bottleneck

```python
# Function to find the maximum bottleneck value between s and t using binary search
def find_max_bottleneck(s, t):
    low = 0
    high = max(edge[2] for edge in edge_list) if edge_list else 0
    ans = 0
    while low <= high:
        mid = (low + high) // 2
        if is_connected(s, t, mid):
            ans = mid
            low = mid + 1
        else:
            high = mid - 1
    return ans
```

- This function uses **binary search** to find the **maximum minimum number of flowers** that can be encountered along a path from `s` to `t`.
- The search is performed on the range of possible flower counts (`w`). For each value `w`, the `is_connected()` function is used to check if there exists a valid path from `s` to `t` with all edges having at least `w` flowers.
- The binary search tries to maximize `w`, and the result is the maximum possible minimum number of flowers for the path.

6. Main Logic

```python
# Process each day and handle the changes
for day in range(q + 1):
    if day > 0:
        # Apply the (day-1)th change
        change = changes[day - 1]
        for (a, b, c) in change:
            idx = find_edge_index(a, b)
            if idx != -1:
                edge_list[idx][2] = c
    # Handle the query for the current day
    s, t = queries[day]
    res = find_max_bottleneck(s, t)
    print(res)
```

- This loop iterates over all days (`q + 1` days).
- For each day:
  - If it's not the first day, the program applies the changes from the previous day to the graph by updating the flower counts of affected edges.
  - It then solves the query for that day by calling `find_max_bottleneck(s, t)` to get the maximum minimum number of flowers for the path between `s` and `t`.
  - The result for each query is printed.

③Why is your solution better than others?

1. Efficient Handling of Dynamic Changes

- The problem involves **dynamic updates** to the graph (edge weights change over time). The binary search approach is well-suited for this because it efficiently adjusts the graph's state each day by applying the changes and then recalculating the query result.
- This is much more efficient than algorithms like **Max-Flow** or **Dijkstra's** which might require full recomputation of paths after every update, leading to higher time complexity.

2. Optimal Search with Binary Search

- By using **binary search** on the possible edge weights, the solution narrows down the maximum possible "bottleneck" (minimum flowers on any edge in the path) in a logarithmic manner. This dramatically reduces the number of checks (from potentially testing all edge weights to just `log(W)` where `W` is the maximum flower count).
- This is far more efficient than methods like **Dijkstra's algorithm** or **Max-Flow** that would need to explore the entire graph for each query.