1782. Count Pairs Of Nodes

Hard Graph Array Two Pointers Binary Search Sorting Leetcode Link

Problem Description

The problem presents an undirected graph with `n` nodes and a list of edges, where each edge `[u_i, v_i]` represents an undirected connection between nodes `u_i` and `v_i`. Along with the graph, there is also an array of queries. The challenge is to compute, for each query, the number of unique pairs of nodes `(a, b)` that satisfy the following conditions:

1. a is less than b (to avoid duplication since the graph is undirected).
2. The number of edges connected to either node `a` or node `b` is greater than the value of the current query.

This is not as straightforward as simply iterating over all pairs of nodes, because the number of possible pairs increases quadratically with the number of nodes, which would result in an inefficient solution.

Intuition

The intuition behind the solution approach is to use a combination of sorting, binary search, and careful counting to efficiently find the number of node pairs `(a, b)` that meet the criteria for each query.

1. Count the incident edges: The first step is to count the number of edges incident to each node. This count is later used to determine if a pair of nodes `(a, b)` has more incidents than the query value.
2. Sort the counts: By sorting the counts of incident edges, we can then use binary search to quickly identify how many nodes have a count of incidents greater or equal to what is needed for a specific query.
3. Binary search to find pairs: For each node `j`, binary search is used to find how many nodes `k` have enough incidents such that the sum of incidents for nodes `j` and `k` is greater than the query value.
4. Adjust for exact matches: Since multiple edges can exist between the same two nodes, we must adjust the pairs count if the exact sum of incidents equals the query value after removing the redundant edges (hence why the adjustment checks if removing the shared edge from the total incidents would fall at or below the query value).

Combining these steps allows us to programmatically calculate the output for all queries in a way that is much more efficient than brute force, enabling the solution to handle larger graphs and queries within acceptable time constraints.

Solution Approach

The solution code implements the following approach:

1. **Initialize Counters**: Initialize a counter list `cnt` with a size equal to the number of nodes. This list will keep track of the number of edges connected to each node. Also, create a dictionary `g`, which will hold the counts of edges between each distinct node pair (used for adjustment later).
2. **Count Edges and Pairs**: Iterate over the list of edges. For each edge `(a, b)`, sort the nodes to ensure that `a < b` (since the graph is undirected) and increase their respective counters in `cnt`. Also, keep a running total of edges between the specific pair in the `g` dictionary.
3. **Sort Node Incidents**: Sort the `cnt` array containing the incidents count for all nodes. This is crucial as it allows us to use binary search in the next step to quickly find eligible pairs for each query.
4. **Processing Queries**: Each query asks how many pairs `(a, b)` there are such that `incident(a, b) > queries[j]`. To answer this, loop through each query in `queries`. For each node count `x` in the sorted `cnt` list:
   - Use binary search `bisect_right` to find the boundary `k` where any node beyond this index `k` in the sorted list when paired with the current node `j`, will have a combined incident count greater than the query value. This effectively counts eligible pairs `(j, k)` for the query, as node `k` to the end of the sorted list would satisfy the query.
   - The difference `n - k` gives the number of possible pairs for that node. Accumulate these values in `ans[i]` for the `i-th` query.
5. **Adjustment for Shared Edges**: After summing up all probable pairs, we must subtract those pairs `(a, b)` where the incident count is only greater than the query after including the shared edges. Thus, for each pair `(a, b)` in the `g` dictionary, if the sum of their individual incidents `cnt[a] + cnt[b]` is greater than the query but no longer greater than the query after subtracting the shared edge count `v`, decrement the answer for that query since we've initially overcounted this pair.
6. **Return Results**: After processing all queries and making all necessary adjustments, return the list `ans` containing the number of valid pairs for each query.

This approach utilizes a combination of counting, sorting, and binary searching to resolve the "pairs with greater incidents than" problem effectively within a polynomial complexity, avoiding a naive quadratic pairing which would be inefficient at scale.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose our graph has `n = 4` nodes and the following list of `edges = [[1, 2], [2, 3], [2, 4], [1, 3]]`. We have an array of queries `queries = [1, 2]` that we need to answer according to the solution approach described above.

1. **Initialize Counters**
   - Initialize the `cnt` array to store the number of edges incident to each node: `cnt = [0, 0, 0, 0]` because we have 4 nodes.
   - Initialize a dictionary `g` to store the counts of edges between each pair of nodes: `g = {}`.
2. **Count Edges and Pairs**
   - For the edge `[1, 2]`, we increment `cnt[0]` and `cnt[1]` by 1 (assuming 1-based indexing for `cnt`: `cnt = [1, 1, 0, 0]`) and add `g[(1, 2)] = 1`.
   - For the edge `[2, 3]`, increment `cnt[1]` and `cnt[2]`, resulting in `cnt = [1, 2, 1, 0]` and `g[(2, 3)] = 1`.
   - For the edge `[2, 4]`, increment `cnt[1]` and `cnt[3]`, leading to `cnt = [1, 3, 1, 1]` and `g[(2, 4)] = 1`.
   - For the edge `[1, 3]`, increment `cnt[0]` and `cnt[2]`, getting `cnt = [2, 3, 2, 1]`, and add `g[(1, 3)] = 1`.
3. **Sort Node Incidents**
   - We sort `cnt` to get `sorted_cnt = [1, 1, 2, 3]`.
4. **Processing Queries**
   - Process each query `q` from `queries`. For `q = 1`, find pairs `(a, b)` where `incident(a, b) > q`. We loop over `sorted_cnt` and use binary search:
     - For node with 1 incident, `bisect_right` finds no other nodes with incidents higher than `q - 1 = 0`, so no pairs are added.
     - For the next node with 1 incident, the same occurs. Still no pairs added.
     - For the node with 2 incidents, `bisect_right` will return index 3 (1-based), meaning there's 1 node satisfying the condition (the node with 3 incidents). We add this pair, and `ans[0] = 1`.
     - For the node with 3 incidents, `bisect_right` will return 4, but since it includes the node itself, we don't count it.
   - For the second query `q = 2`, repeat the process:
     - For nodes with 1 incident, `bisect_right` now finds indices 3 and 4 as `q - 1 = 1`, so two more pairs for `ans[1]`.
     - For the node with 2 incidents, `bisect_right` finds index 4, so one more pair, and we have `ans[1] = 3`.
5. **Adjustment for Shared Edges**
   - There are no shared edges, so no adjustments are needed in this example.
6. **Return Results**
   - After processing all queries, we have the results for the queries. The answer to the query `q = 1` is `ans[0] = 1` and for `q = 2` is `ans[1] = 3`.
   - Return `ans = [1, 3]`.

Given our small graph and queries, the returned answers indicate there is one pair of nodes with more than one incident edge for the first query and three pairs for the second query.

Python Solution

```python
from collections import defaultdict
from bisect import bisect_right
from typing import List

class Solution:
    def countPairs(self, n: int, edges: List[List[int]], queries: List[int]) -> List[int]:
        # Initialize a list to store the count of edges each node is connected to
        edge_count = [0] * n
        # Dictionary to store the count of shared edges between pairs of nodes
        shared_edges_count = defaultdict(int)

        # Calculate the edge count and shared edges count
        for a, b in edges:
            # Decrement to convert to 0-indexed and identify the smaller node
            a, b = a - 1, b - 1
            a, b = min(a, b), max(a, b)
            edge_count[a] += 1
            edge_count[b] += 1
            shared_edges_count[(a, b)] += 1

        # Sort the edge counts
        sorted_edge_count = sorted(edge_count)
        # Initialize the answer list for each query
        answer = [0] * len(queries)

        # Process each query
        for i, threshold in enumerate(queries):
            # For each node in sorted order, count nodes with enough edges to exceed the threshold
            for j, edge_cnt in enumerate(sorted_edge_count):
                # Find the rightmost value greater than the remaining threshold to satisfy the query
                k = bisect_right(sorted_edge_count, threshold - edge_cnt, lo=j + 1)
                # Add the number of nodes with enough edges to the answer
                answer[i] += n - k
            # Adjust the answer for shared edges
            for (a, b), shared_edges in shared_edges_count.items():
                if edge_count[a] + edge_count[b] > threshold and edge_count[a] + edge_count[b] - shared_edges <= threshold:
                    answer[i] -= 1
        return answer
```

Java Solution

```java
class Solution {
    // Method to count pairs based on given queries
    public int[] countPairs(int n, int[][] edges, int[] queries) {
        // Degree count for each node
        int[] degreeCount = new int[n];
        // Map to store the number of shared edges between nodes
        Map<Integer, Integer> sharedEdges = new HashMap<>();

        // Count the degrees and shared edges
        for (int[] edge : edges) {
            int a = edge[0] - 1;
            int b = edge[1] - 1;
            degreeCount[a]++;
            degreeCount[b]++;
            int key = Math.min(a, b) * n + Math.max(a, b);
            sharedEdges.merge(key, 1, Integer::sum);
        }

        // Sort the degrees for binary searching later
        int[] sortedDegrees = degreeCount.clone();
        Arrays.sort(sortedDegrees);

        // Answer array to store result for each query
        int[] answer = new int[queries.length];

        // Process each query
        for (int i = 0; i < queries.length; i++) {
            int queryThreshold = queries[i];

            // Two pointers approach to find valid pairs
            for (int j = 0; j < n; j++) {
                int currentValue = sortedDegrees[j];
                int k = search(sortedDegrees, queryThreshold - currentValue, j + 1);
                answer[i] += n - k;
            }

            // Adjust answer for edges that were counted twice
            for (var entry : sharedEdges.entrySet()) {
                int a = entry.getKey() / n;
                int b = entry.getKey() % n;
                int commonEdges = entry.getValue();

                // If the actual pair was counted, remove it if it shouldn't be
                if (degreeCount[a] + degreeCount[b] > queryThreshold
                    && degreeCount[a] + degreeCount[b] - commonEdges <= queryThreshold) {
                    answer[i]--;
                }
            }
        }
        return answer;
    }

    // Helper method for binary search to find the right position
    private int search(int[] arr, int x, int start) {
        int left = start, right = arr.length;
        while (left < right) {
            int mid = (left + right) / 2;
            if (arr[mid] > x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    vector<int> countPairs(int n, vector<vector<int>>& edges, vector<int>& queries) {
        vector<int> nodeDegree(n); // This vector holds the degree of each node.
        unordered_map<int, int> sharedEdgesCount; // This map will hold the number of shared edges between pairs of nodes.

        // Go through each edge and update degree count and shared edges for the pair.
        for (auto& edge : edges) {
            int node1 = edge[0] - 1;
            int node2 = edge[1] - 1;
            ++nodeDegree[node1];
            ++nodeDegree[node2];
            int combinedKey = min(node1, node2) * n + max(node1, node2); // Unique key for each node pair.
            ++sharedEdgesCount[combinedKey];
        }

        vector<int> sortedDegrees = nodeDegree; // We'll need a sorted version of the degrees for two-pointer technique.
        sort(sortedDegrees.begin(), sortedDegrees.end());

        vector<int> answer(queries.size()); // This will hold our final answer for each query.

        // For each query, count the valid pairs.
        for (int i = 0; i < queries.size(); ++i) {
            int threshold = queries[i];

            // Using the two-pointer technique to find valid pairs by degree sum.
            for (int j = 0; j < n; ++j) {
                int degree = sortedDegrees[j]; // Choose a starting degree from sorted array.
                // Find the position of the smallest element that, when added to `degree`,
                // would exceed `threshold`. Subtract from total nodes to get count.
                int pairsCount = upper_bound(sortedDegrees.begin() + j + 1, sortedDegrees.end(), threshold - degree) - sortedDegree
                answer[i] += n - pairsCount;
            }

            // Adjust answer based on shared edges between node pairs.
            for (auto& [combinedKey, sharedEdges] : sharedEdgesCount) {
                int node1 = combinedKey / n;
                int node2 = combinedKey % n;
                // If sum of degrees of nodes exceeds threshold but subtracting shared edges doesn't,
                // we've previously counted this as a valid pair incorrectly and must decrement.
                if (nodeDegree[node1] + nodeDegree[node2] > threshold && nodeDegree[node1] + nodeDegree[node2] - sharedEdges <= thr
                    --answer[i];
                }
            }
        }
        return answer; // Return the final counts of valid pairs for each query.
    }
};
```

Typescript Solution

```typescript
function countPairs(nodeCount: number, graphEdges: number[][], queryValues: number[]): number[] {
    // Initialize counter for each node with zero
    const edgeCounts: number[] = new Array(nodeCount).fill(0);

    // Map for counting shared edges
    const sharedEdgeCounts: Map<number, number> = new Map();

    // Fill edge counts and shared edges
    for (const [node1, node2] of graphEdges) {
        edgeCounts[node1 - 1]++;
        edgeCounts[node2 - 1]++;
        const key = Math.min(node1 - 1, node2 - 1) * nodeCount + Math.max(node1 - 1, node2 - 1);
        sharedEdgeCounts.set(key, (sharedEdgeCounts.get(key) || 0) + 1);
    }

    // Sort edge counts in ascending order
    const sortedEdgeCounts = [...edgeCounts].sort((a, b) => a - b);

    // Binary search utility function
    const binarySearch = (values: number[], target: number, left: number): number => {
        let right = values.length;
        while (left < right) {
            const mid = (left + right) >> 1;
            if (values[mid] > target) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    };

    // Initialize the resultant array
    const results: number[] = [];

    // For each query, calculate the number of valid pairs
    for (const threshold of queryValues) {
        let pairCount = 0;
        for (let index = 0; index < sortedEdgeCounts.length; ++index) {
            const searchResult = binarySearch(sortedEdgeCounts, threshold - sortedEdgeCounts[index], index + 1);
            pairCount += nodeCount - searchResult;
        }

        // Adjust the count for shared edges
        for (const [key, sharedCount] of sharedEdgeCounts) {
            const node1 = Math.floor(key / nodeCount);
            const node2 = key % nodeCount;
            if (edgeCounts[node1] + edgeCounts[node2] > threshold && edgeCounts[node1] + edgeCounts[node2] - sharedCount <= thresho
                --pairCount;
            }
        }

        // Add the final result for this query
        results.push(pairCount);
    }

    // Return the array of results for all queries
    return results;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code involves several parts:

1. **Creating and populating the graph and count array**: Iterating over each edge to populate the `cnt` and `g` has a time complexity of `O(E)` where `E` is the number of edges.
2. **Sorting the `cnt` array**: Sorting the array of nodes' degrees has a time complexity of `O(N log N)` where `N` is the number of nodes.
3. **Processing queries**: For each query, the code iterates over each possible `x` in `s` (the sorted `cnt` array) and then performs a binary search to find `k` which has a time complexity of `O(log N)`. Since this is inside a loop that goes through `s`, the complexity of this part becomes `O(N log N)`.
4. **Adjusting count for each query based on the graph `g` dictionary**: This step involves iterating over each item in `g` (which would be at most `E`, the number of unique edges) for each query, giving it a complexity of `O(Q * E)` where `Q` is the number of queries.

Combining these parts, the overall time complexity is `O(E) + O(N log N) + O(Q * (N log N + E))`, which simplifies to `O(Q * (N log N + E))` assuming `Q * N log N` dominates `E` and `Q * E` dominates `N log N`.

Space Complexity

The space complexity of the code is influenced by the following components:

1. **The `cnt` array**: Requires `O(N)` space.
2. **The `g` graph representation**: In the worst case, it stores all the unique edges, so it takes `O(E)` space.
3. **The `s` array**: It’s a sorted list of node degrees, which also requires `O(N)` space.
4. **The `ans` array**: This requires `O(Q)` space, where `Q` is the number of queries.
5. **Auxiliary space for sorting**: Sorting an array in Python requires `O(N)` space.

Therefore, the overall space complexity is `O(N + E + Q + N)`, simplifying to `O(N + E + Q)` when not considering the coefficients.