1724. Checking Existence of Edge Length Limited Paths II

Leetcode Link

Problem explanation

Given an undirected graph of `n` nodes. The graph is defined by an `edgeList`, where `edgeList[i] = [ui, vi, disi]` denotes an edge between nodes `ui` and `vi` with a distance `disi`. Note that there may be multiple edges between two nodes, and the graph may not be connected. We need to implement two methods in a class called `DistanceLimitedPathsExist`:

1. `DistanceLimitedPathsExist(n, edgeList)`: Initialize the class with an undirected graph.
2. `query(p, q, limit)`: Returns `true` if there exists a path from `p` to `q` such that each edge on the path has a distance strictly less than `limit`. Otherwise, return `false`.

Example

For example, let's consider the following graph:

```
Input:
n = 6
edgeList = [[0, 2, 4], [0, 3, 2], [1, 2, 3], [2, 3, 1], [4, 5, 5]]
```

We have 6 nodes (`n = 6`), and the graph looks like this:

```
      4
  0 ----- 2
  | \     |
 2|  \1   |3
  |/     \|
  3 ----- 1
```

We want to perform the following queries:

```
query(2, 3, 2) -> true  (There is an edge from 2 to 3 of distance 1, which is less than 2)
query(1, 3, 3) -> false (There is no way to go from 1 to 3 with distances strictly less than 3)
query(2, 0, 3) -> true  (There is a way to go from 2 to 0 with distance < 3: travel from 2 to 3 to 0)
query(0, 5, 6) -> false (There are no paths from 0 to 5)
```

Approach

We can implement a Union-Find data structure to keep track of connected components for different distance limits. The Union-Find structure will have a vector of maps `id`. Each key-value pair in the map at index i represents the maximum distance for which node i is connected.

1. Initialize the Union-Find structure with `n` nodes.
2. Sort the `edgeList` in ascending order by distance.
3. For each edge in the sorted `edgeList`, perform the union operation on the nodes `u` and `v` with the given distance `d`.
4. When query is called with nodes `p` and `q`, and distance `limit`, find the root of `p` and `q` in the Union-Find structure for the given distance limit. If the roots are the same, return `true`. Otherwise, return `false`.

Now let's write the solution for the problem in Python, Java, JavaScript, C++, and C#.

Python Solution

```python
from collections import defaultdict
from sortedcontainers import SortedDict

class UnionFind:
    def __init__(self, n):
        self.id = [SortedDict({0: i}) for i in range(n)]

    def union(self, u, v, limit):
        i = self.find(u, limit)
        j = self.find(v, limit)
        if i == j:
            return
        self.id[i][limit] = j

    def find(self, u, limit):
        it = self.id[u].irange(0, limit)
        i = next(it, u)
        if i == u:
            return u
        j = self.find(i, limit)
        self.id[u][limit] = j
        return j

class DistanceLimitedPathsExist:
    def __init__(self, n, edgeList):
        self.uf = UnionFind(n)
        edgeList.sort(key=lambda x: x[2])
        for u, v, d in edgeList:
            self.uf.union(u, v, d)

    def query(self, p, q, limit):
        return self.uf.find(p, limit - 1) == self.uf.find(q, limit - 1)
```

Java Solution

```java
import java.util.*;

class UnionFind {
    private List<Map<Integer, Integer>> id;

    public UnionFind(int n) {
        id = new ArrayList<>();
        for (int i = 0; i < n; ++i) {
            id.add(i, new TreeMap<>());
            id.get(i).put(0, i);
        }
    }

    public void union(int u, int v, int limit) {
        int i = find(u, limit);
        int j = find(v, limit);
        if (i == j)
            return;
        id.get(i).put(limit, j);
    }

    public int find(int u, int limit) {
        Map.Entry<Integer, Integer> entry = id.get(u).lowerEntry(limit);
        int i = (entry == null) ? u : entry.getValue();
        if (i == u) {
            return u;
        }
        int j = find(i, limit);
        id.get(u).put(limit, j);
        return j;
    }
}

class DistanceLimitedPathsExist {
    private UnionFind uf;

    public DistanceLimitedPathsExist(int n, int[][] edgeList) {
        uf = new UnionFind(n);
        Arrays.sort(edgeList, (a, b) -> Integer.compare(a[2], b[2]));
        for (int[] edge : edgeList) {
            uf.union(edge[0], edge[1], edge[2]);
        }
    }

    public boolean query(int p, int q, int limit) {
        return uf.find(p, limit - 1) == uf.find(q, limit - 1);
    }
}
```

JavaScript Solution

```javascript
class UnionFind {
  constructor(n) {
    this.id = new Array(n).fill(null).map((_, i) => new Map([[0, i]]));
  }

  union(u, v, limit) {
    const i = this.find(u, limit);
    const j = this.find(v, limit);
    if (i === j) {
      return;
    }
    this.id[i].set(limit, j);
  }

  find(u, limit) {
    const it = this.id[u].entries().next();
    const i = it.done || it.value[0] >= limit ? u : it.value[1];
    if (i === u) {
      return u;
    }
    const j = this.find(i, limit);
    this.id[u].set(limit, j);
    return j;
  }
}

class DistanceLimitedPathsExist {
  constructor(n, edgeList) {
    this.uf = new UnionFind(n);

    edgeList.sort((a, b) => a[2] - b[2]);
    for (const [u, v, d] of edgeList) {
      this.uf.union(u, v, d);
    }
  }

  query(p, q, limit) {
    return this.uf.find(p, limit - 1) === this.uf.find(q, limit - 1);
  }
}
```

C++ Solution

```cpp
#include <algorithm>
#include <map>
#include <vector>

using namespace std;

class UnionFind {
 public:
  UnionFind(int n) {
    id.resize(n);
    for (int i = 0; i < n; ++i) id[i][0] = i;
  }

  void union_(int u, int v, int limit) {
    int i = find(u, limit);
    int j = find(v, limit);
    if (i == j) return;
    id[i][limit] = j;
  }

  int find(int u, int limit) {
    auto it = id[u].upper_bound(limit);
    int i = prev(it)->second;
    if (i == u) return u;
    int j = find(i, limit);
    id[u][limit] = j;
    return j;
  }

 private:
  vector<map<int, int>> id;
};

class DistanceLimitedPathsExist {
 public:
  DistanceLimitedPathsExist(int n, vector<vector<int>>& edgeList) {
    uf = UnionFind(n);
    sort(edgeList.begin(), edgeList.end(),
         [](const auto& a, const auto& b) { return a[2] < b[2]; });

    for (const vector<int>& edge : edgeList) {
      int u = edge[0];
      int v = edge[1];
      int d = edge[2];
      uf.union_(u, v, d);
    }
  }

  bool query(int p, int q, int limit) {
    return uf.find(p, limit - 1) == uf.find(q, limit - 1);
  }

 private:
  UnionFind uf;
};
```

C# Solution

```csharp
using System;
using System.Collections.Generic;
using System.Linq;

public class UnionFind {
    private List<SortedDictionary<int, int>> id;

    public UnionFind(int n) {
        id = new List<SortedDictionary<int, int>>(n);
        for (int i = 0; i < n; ++i) {
            id.Add(new SortedDictionary<int, int>());
            id[i][0] = i;
        }
    }

    public void Union(int u, int v, int limit) {
        int i = Find(u, limit);
        int j = Find(v, limit);
        if (i == j) {
            return;
        }
        id[i][limit] = j;
    }

    public int Find(int u, int limit) {
        KeyValuePair<int, int> entry;
        var it = id[u].Reverse().FirstOrDefault(x => x.Key < limit);
        entry = it;
        int i = (entry.Key == 0 && entry.Value == 0) ? u : entry.Value;
        if (i == u) {
            return u;
        }
        int j = Find(i, limit);
        id[u][limit] = j;
        return j;
    }
}

public class DistanceLimitedPathsExist {
    private UnionFind uf;

    public DistanceLimitedPathsExist(int n, int[][] edgeList) {
        uf = new UnionFind(n);
        Array.Sort(edgeList, (a, b) => a[2].CompareTo(b[2]));
        foreach (int[] edge in edgeList) {
            uf.Union(edge[0], edge[1], edge[2]);
        }
    }

    public bool Query(int p, int q, int limit) {
        return uf.Find(p, limit - 1) == uf.Find(q, limit - 1);
    }
}
```

In this problem, we have been able to implement a data structure called `DistanceLimitedPathsExist` that can process queries on a given undirected graph to check if paths exist between two nodes having each edge's distance strictly less than a given limit. To solve this problem, we have used the Union-Find data structure and have been able to implement the solution in Python, Java, JavaScript, C++, and C# languages. This data structure can be quite useful in solving problems related to querying connected components and their properties in a graph.

Got a question? Ask the Teaching Assistant anything you don't understand.