其实以下解法与其说是prim不如说是kruskal，因为实际上是最优先按照cost排序的，而不是最优先任意选择一个节点开始的，这样更符合kruskal的定义。

更像prim方法的解法：

<https://blog.csdn.net/roufoo/article/details/85271791>

更像kruskal方法的解法：

<http://www.noteanddata.com/lintcode-629-Minimum-Spanning-Tree.html>

<http://buttercola.blogspot.com/2019/03/lintcode-629-minimum-spanning-tree.html>

<https://www.cnblogs.com/lz87/p/7476882.html>

[[LintCode] Minimum Spanning Tree](https://www.cnblogs.com/lz87/p/7476882.html)

Given a list of Connections, which is the Connection class (the city name at both ends of the edge and a cost between them), find some edges, connect all the cities and spend the least amount.  
Return the connects if can connect all the cities, otherwise return empty list.

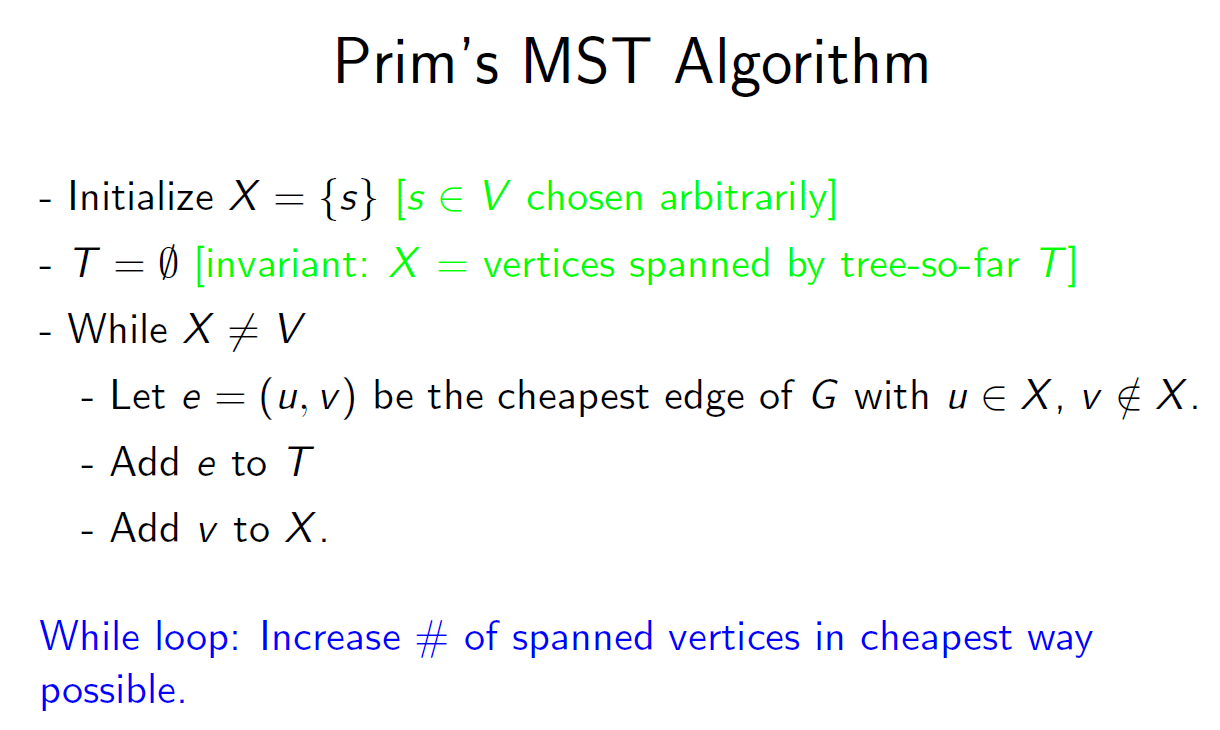
Return the connections sorted by the cost, or sorted city1 name if their cost is same, or sorted city2 if their city1 name is also same.

**Example**

Gievn the connections = ["Acity","Bcity",1], ["Acity","Ccity",2], ["Bcity","Ccity",3]

Return ["Acity","Bcity",1], ["Acity","Ccity",2]

This problem is the classical minimum spanning tree problem with the following Prim's MST Algorithm ready to be used.



Solution 1. Union Find with Prim's Greedy MST algorithm.

The core idea of prim's MST algorithm is that as long as we have not spanned all vertices(cities), we keep picking the cheapest edge e = (u, v), u is in X and v is NOT in X.

We can use a union find data structure to simulate this process. If we pick an edge(connection), we would have the following 2 cases.

a. city1 and city2 are already connected: this means this edge does not satisfy the condition that city1 is in X and city2 is NOT in X, this connection should be ignored.

b. city1 and city2 are not connected: this means this edge satisfies the above condition, if it is the cheapst edge out of all those edges that meets the condition, we should

select this connection and add it to the final result.

Based on the above analysis, we have the following algorithm.

1. sort the connections to make sure smaller cost connections are in front. (Prim MST greedy)

2. create a mapping between city names and union find index as it is best to use integer as union find's index.

　use numbers from 0 to n - 1 for all cities assuming there are n different cities.

　As we are creating the mapping, the next available integer index also represents the total number of cities whose mapping are created so far. After all the mapping is done,

    this idx variable tells us the total number of nodes(cities). This is important in checking if there is any city that is disconnected with all other cities.

    Based on connected graph theory, if all n cities are connected, there we would include n - 1 different edges that do not introduce any cycle, thus generating the MST.

    However, if there is a part of the graph that are disconnected from the rest, then it means we must only included fewer than n - 1 different connections, otherwise all cities would be connected.

3. Iterate all connections, add to the final result each connection whose ends are not connected in the uf and connect both ends' mapping in the uf.

4. Check if there are n - 1 connections in the final result. If there aren't return an empty list to indicate there are disconnected cities in the given connections.

Runtime/Space complexity: O(m \* logm + 2 \* m + n) ~ O(m \* logm) runtime; O(n) space

Assuming there are n different cities and m different edges,

1. sorting: O(m \* log m) runtime, O(1) space assuming in place quick sort is used.

2. mapping: O(m) runtime, O(n) space.

3. unionfind creation: O(n) runtime, O(n)  space.

4. connections iteration: O(m) runtime, as both the uf find and connect operations take O(1) time on average.

/\*\*

\* Definition for a Connection.

\* public class Connection {

\* public String city1, city2;

\* public int cost;

\* public Connection(String city1, String city2, int cost) {

\* this.city1 = city1;

\* this.city2 = city2;

\* this.cost = cost;

\* }

\* }

\*/

class UnionFind {

int[] father;

UnionFind(int n) {

father = new int[n];

for(int i = 0; i < n; i++) {

father[i] = i;

}

}

int find(int x) {

if(father[x] == x) {

return x;

}

return father[x] = find(father[x]);

}

void connect(int a, int b) {

int root\_a = find(a);

int root\_b = find(b);

if(root\_a != root\_b) {

father[root\_a] = root\_b;

}

}

}

public class Solution {

private Comparator<Connection> comp = new Comparator<Connection>() {

public int compare(Connection c1, Connection c2) {

if(c1.cost != c2.cost) {

return c1.cost - c2.cost;

}

else if(!c1.city1.equals(c2.city1)) {

return c1.city1.compareTo(c2.city1);

}

return c1.city2.compareTo(c2.city2);

}

};

public List<Connection> lowestCost(List<Connection> connections) {

List<Connection> mst = new ArrayList<Connection>();

if(connections == null || connections.size() == 0) {

return mst;

}

Collections.sort(connections, comp);

int idx = 0;

HashMap<String, Integer> strToIdxMap = new HashMap<String, Integer>();

for(Connection c : connections) {

if(!strToIdxMap.containsKey(c.city1)) {

strToIdxMap.put(c.city1, idx++);

}

if(!strToIdxMap.containsKey(c.city2)) {

strToIdxMap.put(c.city2, idx++);

}

}

UnionFind uf = new UnionFind(idx);

for(Connection c : connections) {

int city1Root = uf.find(strToIdxMap.get(c.city1));

int city2Root = uf.find(strToIdxMap.get(c.city2));

if(city1Root != city2Root) {

mst.add(c);

uf.connect(city1Root, city2Root);

}

}

if(mst.size() < idx - 1) {

return new ArrayList<Connection>();

}

return mst;

}

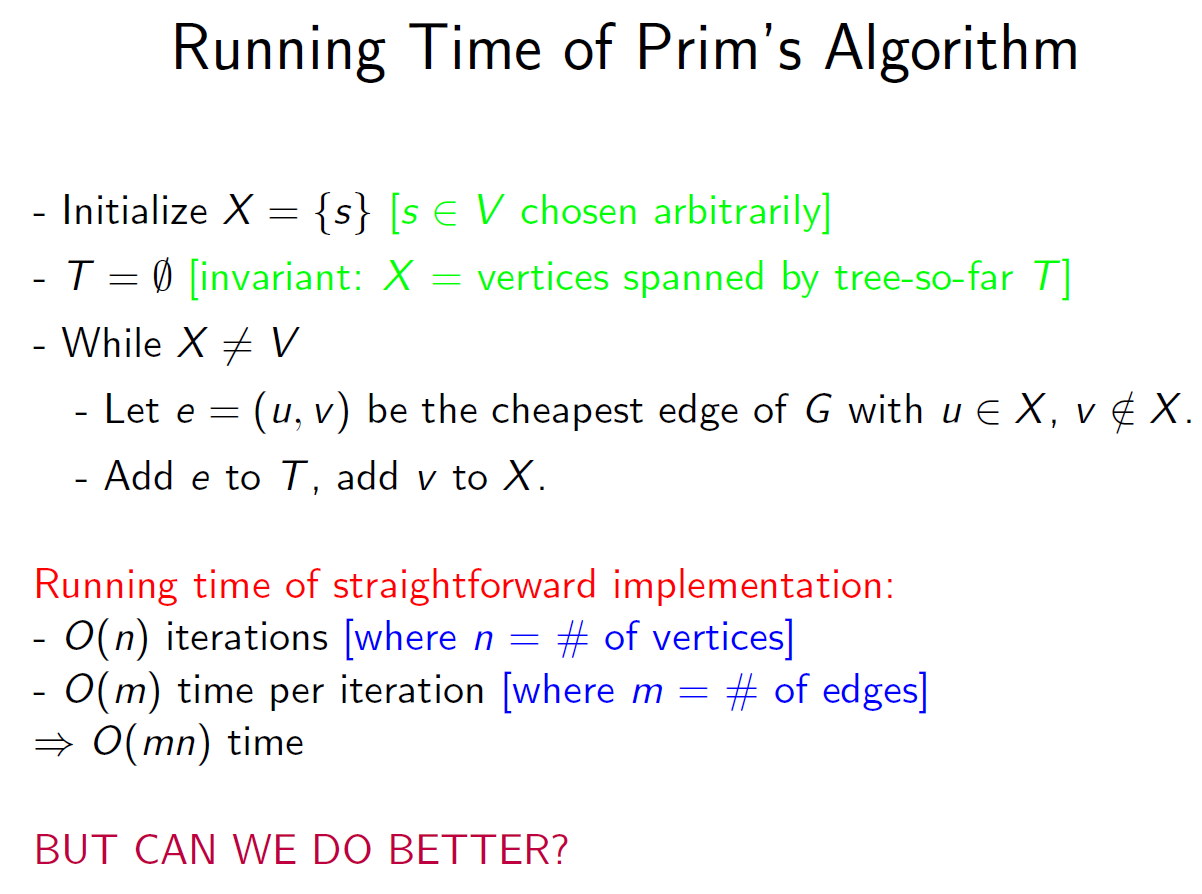
}

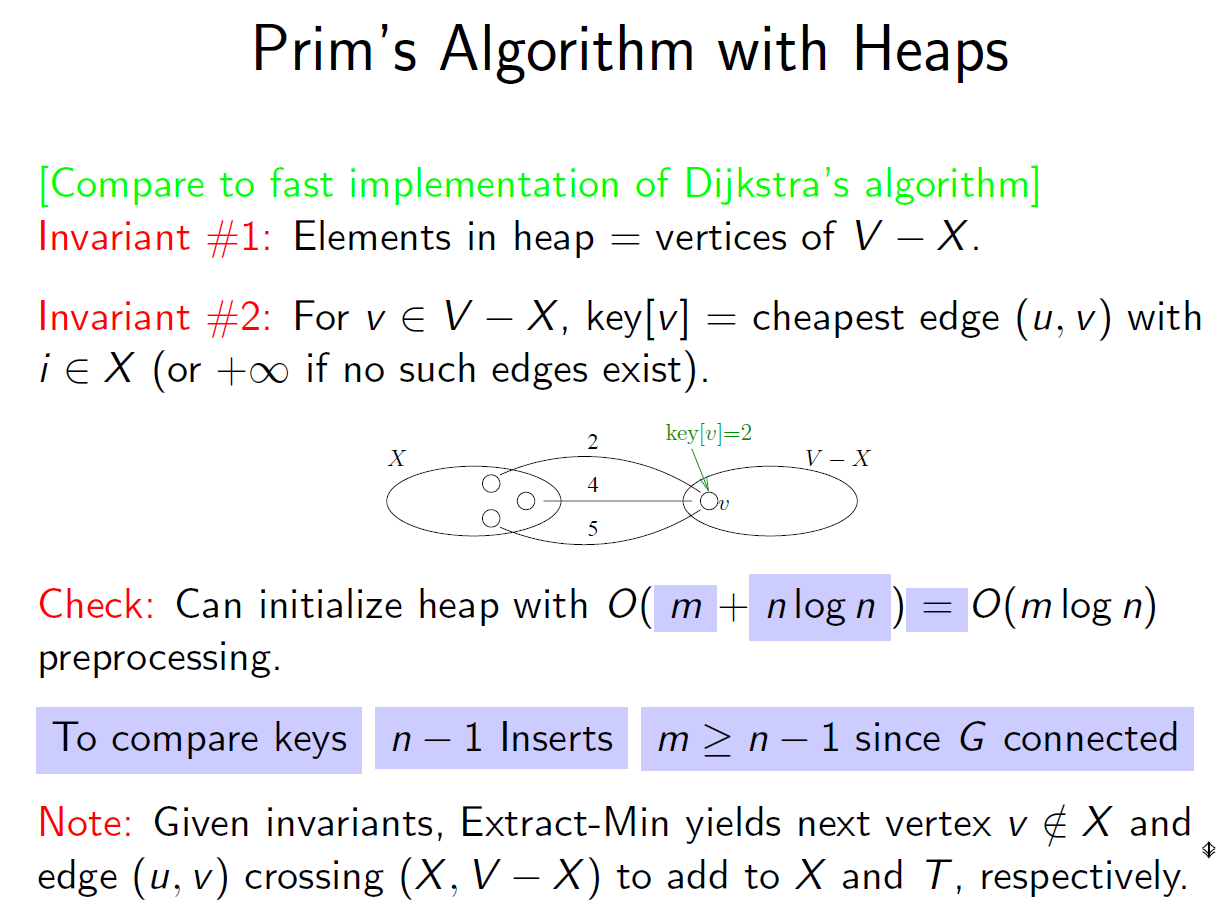
Solution 2.  Prim's Greedy MST algorithm with Adjacent List Graph Representation and Priority Queue.

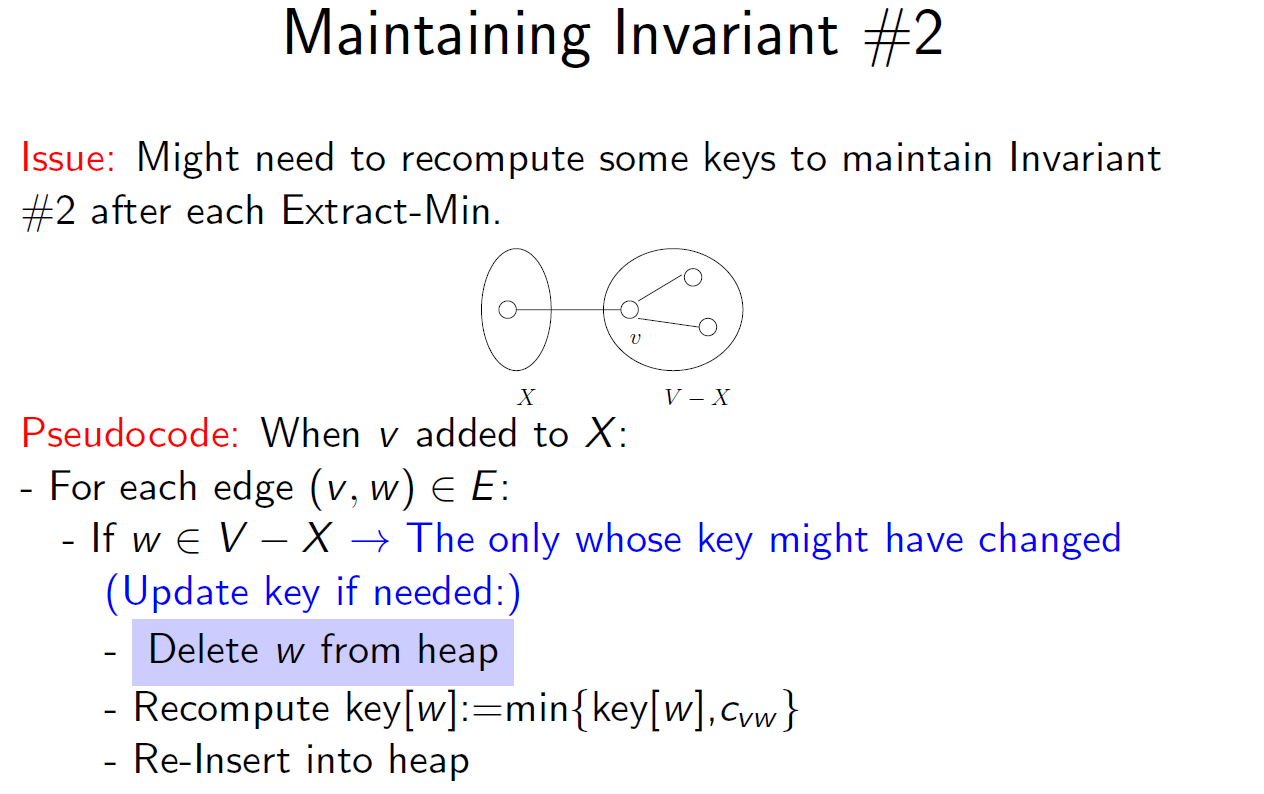
This solution gives O(m \* log n) runtime and O(n + m) space for converting the input connections into a graph representation and the usage of heap.

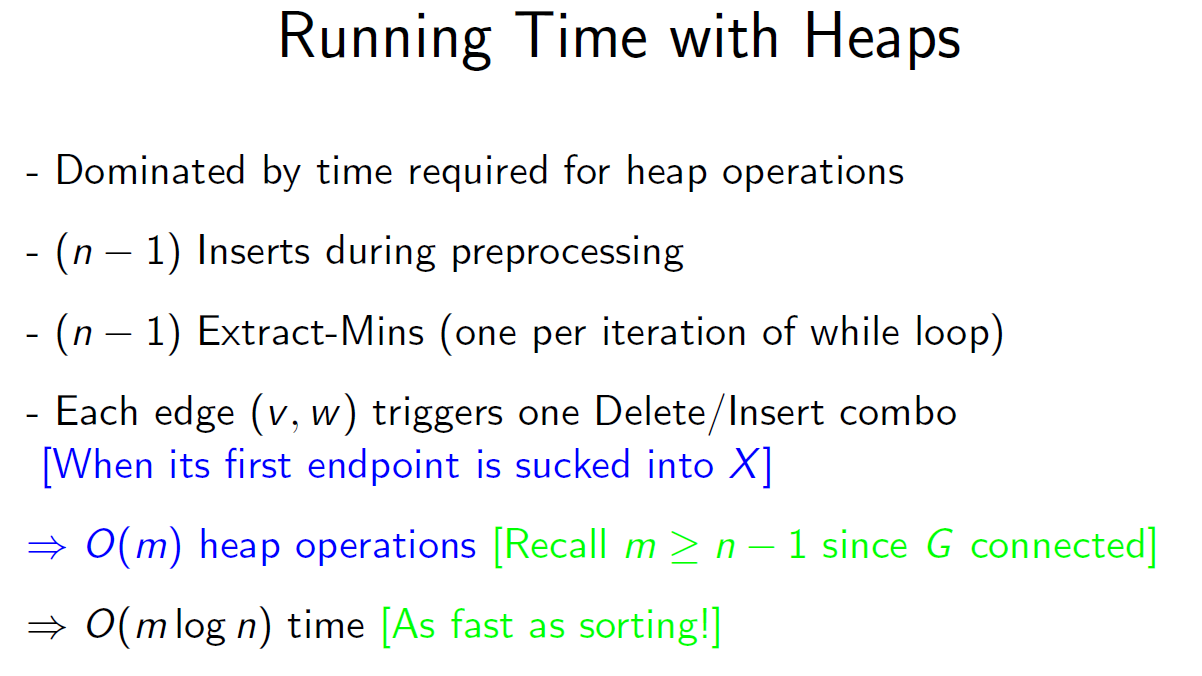
However, this solution is not as good as solution 1 in that extra preprocessing(list of connections to graph) and postprocessing(converting mst to a list of sorted collections as the required output)

are needed.









This implementation uses heap data structure which should supports O(logn) aribitrary removal operation given that it already know the reference to the to be removed node. The also needs to support O(1) look up for checking if a vertex is in the heap or not.

更像prim方法的解法：

<https://blog.csdn.net/roufoo/article/details/85271791>

Prim。我的方法是先将City按名称排序，先取第一个放入joint set，然后更新每个节点跟joint set的距离，再看那个节点离joint set 最近，将该节点放入joint set，同时更新每个节点跟joint set的距离，如此反复，直到所有节点都放入了joint set。

但我的solution不能pass，我看了一下fail的case，我的方案的output跟expected基本一样，只有一个distance不同 ，但distance的cost是一样的。这个可能是因为MST不唯一。

顺便提一下，什么样的图的MST唯一呢? 根据

<https://zhidao.baidu.com/question/509521548.html>

当带权连通图的任意一个环中所包含的权值均不相同，其MST是唯一的。

先正常求出最小生成树，再求次小生成树（具体可以枚举图上其他边加到树里，同时删去重复的边，找到权值和最小的删边方法），如果求出次小生成树的权值和与最小生成树不相等，则最小生成树唯一，否则不唯一。

/\*\*

\* Definition for a Connection.

\* class Connection {

\* public:

\* string city1, city2;

\* int cost;

\* Connection(string& city1, string& city2, int cost) {

\* this->city1 = city1;

\* this->city2 = city2;

\* this->cost = cost;

\* }

\*/

bool operator < (const Connection& c1, const Connection& c2) {

if (c1.cost != c2.cost)

return c1.cost < c2.cost;

if (c1.city1 != c2.city1)

return c1.city1 < c2.city1;

return c1.city2 < c2.city2;

}

class Solution {

public:

/\*\*

\* @param connections given a list of connections include two cities and cost

\* @return a list of connections from results

\*/

vector<Connection> lowestCost(vector<Connection>& connections) {

// MST - Prime algorithm

vector<Connection> result;

if (connections.size() == 0) return result;

sort(connections.begin(), connections.end());

set<string> nameSet;

for (auto c : connections) {

nameSet.insert(c.city1);

nameSet.insert(c.city2);

}

int len = nameSet.size();

vector<string> nameVec(nameSet.begin(), nameSet.end());

map<string, int> nameMap; //city\_name, city\_index

int index = 0;

for (auto s : nameSet) {

nameMap[s] = index++;

}

string dummy = "INVALID";

vector<vector<Connection>> graph(len, vector<Connection>(len, Connection(dummy, dummy, INT\_MAX))); //need to do this because Connection does not provide default construction

for (int i = 0; i < len; ++i) {

for (int j = 0; j < len; ++j) {

graph[i][j] = Connection(nameVec[i], nameVec[j], INT\_MAX);

}

}

for (auto c : connections) {

if ((c < graph[nameMap[c.city1]][nameMap[c.city2]]) &&

(c < graph[nameMap[c.city2]][nameMap[c.city1]])) {

graph[nameMap[c.city1]][nameMap[c.city2]] = c;

graph[nameMap[c.city2]][nameMap[c.city1]] = c;

}

}

vector<bool> visited(len, false);

visited[0] = true;

map<int, Connection \*> dist; //city\_index, distance to mst

for (int i = 0; i < len; ++i) {

dist[i] = &graph[0][i];

}

for (int i = 1; i < len; ++i) {

int minDist = INT\_MAX;

for (int j = 0; j < len; ++j) {

if (!visited[j] && dist[j]->cost < minDist) {

minDist = dist[j]->cost;

index = j;

}

}

if (minDist == INT\_MAX) return vector<Connection>();

visited[index] = true;

result.push\_back(\*dist[index]);

for (int j = 0; j < len; ++j) {

if (!visited[j]) {

if (dist[j]->cost > graph[index][j].cost) dist[j] = &graph[index][j];

}

}

}

sort(result.begin(), result.end());

return result;

}

};

更像kruskal方法的解法：

<http://www.noteanddata.com/lintcode-629-Minimum-Spanning-Tree.html>

## [**题目 Lintcode 629 Minimum Spanning Tree**](http://www.noteanddata.com/lintcode-629-Minimum-Spanning-Tree.html#%E9%A2%98%E7%9B%AE-lintcode-629-minimum-spanning-tree)

1. 输入一些connection， 每个connection包括city1, city2, cost, 表示两个城市之间连接的成本
2. 要求返回一个connection的列表， 使得可以连接到所有的城市，并且成本最小
3. 返回结果要排序， 按cost, city1, city2排序

## [**解题思路分析**](http://www.noteanddata.com/lintcode-629-Minimum-Spanning-Tree.html#%E8%A7%A3%E9%A2%98%E6%80%9D%E8%B7%AF%E5%88%86%E6%9E%90)

1. 这个就是标准的最小生成树， 就是在图上取一些边，使得整个树不存在环， 然后
2. 最小生成树有好几种做法， 这里采用Kruskal算法，就是先排序， 每次都从成本最小的开始选， 如果不会够成环，就加到结果里面去，  
   直到遍历到够了（也就是边的个数=节点个数-1）。判断环的地方，用union find算法
3. 这里做的时候出现一些错误，比如要连接所有节点的最小生成树可能不存在，所以循环退出的条件一开始搞错了
4. 然后这个题目要求结果是排序的, 一开始排序也没有对

### [**时间复杂度**](http://www.noteanddata.com/lintcode-629-Minimum-Spanning-Tree.html#%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6)

O(N\*logN), N是list的大小， 主要的复杂度是对每条边做循环， 然后每次需要做priorityqueue.poll()操作

### [**空间复杂度**](http://www.noteanddata.com/lintcode-629-Minimum-Spanning-Tree.html#%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6)

O(N+V), N是list的大小， v是里面节点的个数

public class Solution {

public List < Connection > lowestCost(List < Connection > connections) {

if (null == connections || 0 == connections.size()) {

return new ArrayList < > ();

}

// Write your code here

Map < String, Integer > cityMap = new HashMap < > ();

PriorityQueue < Connection > pq = new PriorityQueue(connections.size(), new Comparator < Connection > () {

public int compare(Connection c1, Connection c2) {

return c1.cost - c2.cost;

}

});

int index = 0;

for (Connection conn: connections) {

if (!cityMap.containsKey(conn.city1)) {

cityMap.put(conn.city1, index++);

}

if (!cityMap.containsKey(conn.city2)) {

cityMap.put(conn.city2, index++);

}

pq.add(conn);

}

UnionFind uf = new UnionFind(cityMap.size());

List < Connection > mst = new ArrayList < > ();

while (pq.size() > 0 && mst.size() < cityMap.size() - 1) {

Connection conn = pq.poll();

int index1 = cityMap.get(conn.city1);

int index2 = cityMap.get(conn.city2);

if (!uf.find(index1, index2)) {

uf.union(index1, index2);

mst.add(conn);

}

}

if (mst.size() < cityMap.size() - 1) {

return new ArrayList < > ();

}

Collections.sort(mst, (Connection conn1, Connection conn2) - > {

if (0 != conn1.cost - conn2.cost) {

return conn1.cost - conn2.cost;

} else if (!conn1.city1.equals(conn2.city1)) {

return conn1.city1.compareTo(conn2.city1);

} else {

return conn1.city2.compareTo(conn2.city2);

}

});

return mst;

}

static class UnionFind {

private int[] ids;

public UnionFind(int n) {

this.ids = new int[n];

for (int i = 0; i < n; ++i) {

this.ids[i] = i;

}

}

public int root(int i) {

while (ids[i] != i) {

i = ids[i];

}

return i;

}

public boolean find(int i, int j) {

return root(i) == root(j);

}

public void union(int i, int j) {

int rooti = root(i);

int rootj = root(j);

ids[rooti] = rootj;

}

}

}

<http://buttercola.blogspot.com/2019/03/lintcode-629-minimum-spanning-tree.html>

**Solution:**  
Classic Kruskal algorithm, which is a greedy algorithm. The key idea is sort the connections by the cost. Then connect all the cities using Union-Find. Since we use the edges with the minimal costs first, and the Union-Find will result in a tree. After we connect all the edges, we will end with a minimum spanning tree.

/\*\*

 \* Definition for a Connection.

 \* public class Connection {

 \*   public String city1, city2;

 \*   public int cost;

 \*   public Connection(String city1, String city2, int cost) {

 \*       this.city1 = city1;

 \*       this.city2 = city2;

 \*       this.cost = cost;

 \*   }

 \* }

 \*/

public class Solution {

    /\*\*

     \* @param connections given a list of connections include two cities and cost

     \* @return a list of connections from results

     \*/

    public List<Connection> lowestCost(List<Connection> connections) {

        List<Connection> ans = new ArrayList<>();

        if (connections == null || connections.size() == 0) {

            return ans;

        }

        Collections.sort(connections, new MyConnectionComparator());

        int count = 0;

        Map<String, Integer> map = new HashMap<>();

        for (Connection connection : connections) {

            if (!map.containsKey(connection.city1)) {

                map.put(connection.city1, count++);

            }

            if (!map.containsKey(connection.city2)) {

                map.put(connection.city2, count++);

            }

        }

        int[] parents = new int[count];

        for (int i = 0; i < count; i++) {

            parents[i] = i;

        }

        for (Connection connection : connections) {

            int rootA = find(map.get(connection.city1), parents);

            int rootB = find(map.get(connection.city2), parents);

            if (rootA != rootB) {

                parents[rootA] = rootB;

                ans.add(connection);

            }

        }

        return ans.size() == count - 1 ? ans : new ArrayList<>();

    }

    private int find(int a, int[] parents) {

        int root = a;

        while (root != parents[root]) {

            root = parents[root];

        }

        while (root != a) {

            int temp = parents[a];

            parents[a] = root;

            a = temp;

        }

        return root;

    }

}

class MyConnectionComparator implements Comparator<Connection> {

    @Override

    public int compare(Connection a, Connection b) {

        if (a.cost != b.cost) {

            return a.cost - b.cost;

        }

        if (!a.city1.equals(b.city1)) {

            return a.city1.compareTo(b.city1);

        }

        return a.city2.compareTo(b.city2);

    }

}