入选理由

- 和昨天的题目类似，但是又不那么直接？毕竟是困难，要点面子不是？

标签

- 并查集
- DFS

难度

- 中等

题目地址（924. 尽量减少恶意软件的传播）

https://leetcode-cn.com/problems/minimize-malware-spread

题目描述

```
在节点网络中，只有当 graph[i][j] = 1 时，每个节点 i 能够直接连接到另一个节点 j。

一些节点 initial 最初被恶意软件感染。只要两个节点直接连接，且其中至少一个节点受到恶意软件的感染，那么两个节点都将被恶意软件感染。这种恶意软件的

假设 M(initial) 是在恶意软件停止传播之后，整个网络中感染恶意软件的最终节点数。

我们可以从初始列表中删除一个节点。如果移除这一节点将最小化 M(initial)， 则返回该节点。如果有多个节点满足条件，就返回索引最小的节点。

请注意，如果某个节点已从受感染节点的列表 initial 中删除，它以后可能仍然因恶意软件传播而受到感染。


示例 1:

输入: graph = [[1,1,0],[1,1,0],[0,0,1]], initial = [0,1]
输出: 0
示例 2:

输入: graph = [[1,0,0],[0,1,0],[0,0,1]], initial = [0,2]
输出: 0
示例 3:

输入: graph = [[1,1,1],[1,1,1],[1,1,1]], initial = [1,2]
输出: 1
```

```
提示：

1 < graph.length = graph[0].length <= 300
0 <= graph[i][j] == graph[j][i] <= 1
graph[i][i] == 1
1 <= initial.length < graph.length
0 <= initial[i] < graph.length
```

思路

这道题抽象一下就是在求联通分量

1. 根据 initial 节点去求联通分量
2. 如果两个 inittial 节点在同一个联通分量，这两个节点肯定不是答案，因为不管排除哪个，这个联通分量的节点都会被感染
3. 统计只含有一个初始节点的联通分量，找到联通分量中节点数最多的即可，如果有多个联通分量节点数最多，返回含有最小下标初始节点

上述过程就是找联通分量过程，并查集天然适合找联通分量。

代码

代码支持：JS，Python,CPP,Java

JS Code：

```
var minMalwareSpread = function (graph, initial) {
  const father = Array.from(graph, (v, i) => i);
  function find(v) {
    if (v === father[v]) {
      return v;
    }
    father[v] = find(father[v]);
    return father[v];
  }
  function union(x, y) {
    if (find(x) !== find(y)) {
      father[x] = find(y);
    }
  }

  for (let i = 0; i < graph.length; i++) {
    for (let j = 0; j < graph[0].length; j++) {
      if (graph[i][j]) {
        union(i, j);
      }
    }
```

```
  }

  initial.sort((a, b) => a - b);

  let counts = graph.reduce((acc, cur, index) => {
    let root = find(index);
    if (!acc[root]) {
      acc[root] = 0;
    }
    acc[root]++;
    return acc;
  }, {});

  let res = initial[0];
  let count = -Infinity;

  initial
    .map((v) => find(v))
    .forEach((item, index, arr) => {
      if (arr.indexOf(item) === arr.lastIndexOf(item)) {
        if (count === -Infinity || counts[item] > count) {
          res = initial[index];
          count = counts[item];
        }
      }
    });

  return res;
};
```

Python Code：

```
class UnionFind:
    def __init__(self):
        self.father = {}
        self.size = {}

    def find(self, x):
        self.father.setdefault(x, x)
        if x != self.father[x]:
            self.father[x] = self.find(self.father[x])
        return self.father[x]

    def union(self, x, y):
        fx, fy = self.find(x), self.find(y)
        if self.size.setdefault(fx, 1) < self.size.setdefault(fy, 1):
            self.father[fx] = fy
            self.size[fy] += self.size[fx]
        elif fx != fy:
            self.father[fy] = fx
            self.size[fx] += self.size[fy]
```

```python
class Solution:
    def minMalwareSpread(self, graph: List[List[int]], initial: List[int]) -> int:
        uf = UnionFind()

        for i in range(len(graph)):
            for j in range(i, len(graph)):
                if graph[i][j]:
                    uf.union(i, j)

        initial.sort()
        max_size, index, fi = 0, -1, []
        cnt = collections.defaultdict(int)
        for init in initial:
            fi.append(uf.find(init))
            cnt[fi[-1]] += 1
        for i in range(len(initial)):
            if cnt[fi[i]] > 1:
                continue
            if uf.size[fi[i]] > max_size:
                max_size = uf.size[fi[i]]
                index = initial[i]

        return index if index != -1 else initial[0]
```

CPP Code:

```cpp
class UF
{
vector<int> parent;
vector<int> weight;
int count;
public:
    UF(int n)
    {
        for(int i=0; i< n; i++)
        {
            parent.push_back(i);
            weight.push_back(1);
        }
        count = n;
    }

    void union(int i, int j)
    {
        int parentI = find(i);
        int parentJ = find(j);
        if(parentI != parentJ)
        {
            if(weight[parentI] < weight[parentJ])
            {
```

```
                parent[parentI] = parentJ;
                weight[parentJ] += weight[parentI];
                weight[parentI]=0 ;
                count --;
            }
            else
            {
                parent[parentJ] = parentI;
                weight[parentI] += weight[parentJ];
                weight[parentJ]=0;
                count--;
            }
        }
    }
    int find(int i)
    {
        while(parent[i]!=i)
        {
            parent[i] = parent[parent[i]];
            i = parent[i];
        }
        return i;
    }
    int getWeight(int i)
    {
        return weight[i];
    }

};
class Solution {
public:
    int minMalwareSpread(vector<vector<int>>& graph, vector<int>& initial) {

        UF uf(graph.size());
        for(int i=0; i< graph.size(); i++)
        {
            for(int j=i+1; j< graph[i].size(); j++)
            {
                if(graph[i][j])
                  uf.union(i, j);
            }
        }

        unordered_map<int, int> record;
        sort(initial.begin(), initial.end());
        int totalEffectNode =0;
        for(int i=0; i< initial.size(); i++)
        {
            int rootNode =uf.find(initial[i]);
            record[rootNode]++;
            if(record[rootNode]==1)
            {
                totalEffectNode += uf.getWeight(rootNode);
            }
        }
```

```
        int minNum =INT_MAX;
        int ret =-1;
        for(int i=0; i< initial.size(); i++)
        {
            int rootNode = uf.find(initial[i]);
            if(record[rootNode]==1)
            {
                int effectNode = totalEffectNode - uf.getWeight(rootNode);
                if(effectNode<minNum)
                {
                    minNum = effectNode; ;
                    ret = initial[i];
                }
            }
            else
            {
                if(totalEffectNode < minNum)
                {
                    minNum = totalEffectNode;
                    ret =initial[i];
                }
            }
        }
        return ret;
    }
};
```

Java Çode:

```
class Solution {

    public int minMalwareSpread(int[][] graph, int[] initial) {
        int n = graph.length;
        UF uf = new UF(n);
        for(int i=0;i<n;i++) {
            for(int j=i+1;j<n;j++) {
                if(graph[i][j]==1) {
                    uf.union(i,j);
                }
            }
        }
        int[] count = new int[n];
        for(int node: initial) {
            count[uf.find(node)]++;
        }
        int ans=-1, ansSize=-1;
        for(int node: initial) {
            int root = uf.find(node);
            if(count[root]==1) {
                int currSize = uf.getSize(root);
                if(currSize>ansSize) {
```

```
                    ansSize=currSize;
                    ans=node;
                }
                else if(currSize == ansSize && node < ans) {
                    ans=node;
                }
            }
        }
        if(ans==-1) {
            ans=n+1;
            for(int node: initial) {
                ans = Math.min(node,ans);
            }
        }
        return ans;

    }
    class UF {
        int[] parent;
        int[] size;

        public UF(int n) {
            parent = new int[n];
            size = new int[n];
            for(int i=0;i<n;i++) {
                parent[i]=i;
                size[i]=1;
            }
        }
        public int find(int x) {
            while(parent[x]!=x) {
                parent[x]=parent[parent[x]];
                x=parent[x];
            }
            return parent[x];
        }
        public int getSize(int x) {
            return size[find(x)];
        }
        public void union(int x, int y) {
            int xroot = find(x);
            int yroot = find(y);
            if(xroot!=yroot) {
                if(size[xroot]>size[yroot]) {
                    parent[yroot]=xroot;
                    size[xroot]+=size[yroot];
                }
                else {
                    parent[xroot]=yroot;
                    size[yroot]+=size[xroot];
                }
            }

        }
```

```
    }
}
```

复杂度分析

令 d 为矩阵 M 的大小。

- 时间复杂度：由于使用了路径压缩和按秩合并，因此时间复杂度为 $O(\log(m \times \text{Alpha}(n)))$，n 为合并的次数， m 为查找的次数，这里 Alpha 是 Ackerman 函数的某个反函数
- 空间复杂度：$O(d)$。

DFS

思路

正如之前所说，能用并查集通常也能用搜索（BFS 或者 DFS）。

使用 DFS 的思路比较常规，就是从每个点启动一次搜索。

代码

代码支持：JS，Python

JS Code：

```
var minMalwareSpread = function (graph, initial) {
  const N = graph.length;
  initial.sort((a, b) => a - b);
  let colors = Array.from({ length: N }).fill(0);
  let curColor = 1;
  // 给联通分量标色
  for (let i = 0; i < N; i++) {
    if (colors[i] === 0) {
      dfs(i, curColor++);
    }
  }

  let counts = Array.from({ length: curColor }).fill(0);
  for (node of initial) {
    counts[colors[node]]++;
  }

  let maybe = [];
  for (node of initial) {
    if (counts[colors[node]] === 1) {
```

```
      maybe.push(node);
    }
  }

  counts.fill(0);

  for (let i = 0; i < N; i++) {
    counts[colors[i]]++;
  }

  let res = -1;
  let maxCount = -1;

  for (let node of maybe) {
    if (counts[colors[node]] > maxCount) {
      maxCount = counts[colors[node]];
      res = node;
    }
  }

  if (res === -1) {
    res = Math.min(...initial);
  }

  return res;

  function dfs(start, color) {
    colors[start] = color;
    for (let i = 0; i < N; i++) {
      if (graph[start][i] === 1 && colors[i] === 0) {
        dfs(i, color);
      }
    }
  }
};
```

Python Code:

```
class Solution:
    def minMalwareSpread(self, graph, initial):
        def dfs(i):
            nodes.add(i)
            for j in range(len(graph[i])):
                if graph[i][j] and j not in nodes:
                    dfs(j)
        rank, initial = collections.defaultdict(list), set(initial)
        for node in sorted(initial):
            nodes = set()
            dfs(node)
            if nodes & initial == {node}:
                rank[len(nodes)].append(node)
        return rank[max(rank)][0] if rank else min(initial)
```

复杂度分析

令 d 为矩阵 M 的大小，e 为 initial 长度。

- 时间复杂度：由于使用了排序，因此排序需要时间为 eloge。而 dfs 部分，由于每个点呗访问最多一次，因此时间为 $O(d^2)$
- 空间复杂度：我们使用了 colors 和 counts，因此空间复杂度为 $O(d)$。

上一页 下一页

© 2020 lucifer. 保留所有权利