Mock Interviews @ ACM

Solution Card

Snowed In

Problem Statement

Snow has blanketed Princeton's campus, and the paths that connect Princeton's buildings to one another have been blocked off by multiple feet of the powdery material. To optimally schedule the shoveling of snow, it is necessary to send someone to check the condition of each path on campus. However, this is time-consuming, so we want to start by clearing off the *critical paths*. A path is critical if its removal will cause any building to be unable to be reached from any other building. Princeton's buildings have been numbered 0 to n - 1: given pairs (a_i, b_i) representing the undirected paths between them, return all critical paths that should be cleared off immediately.

Sources: https://leetcode.com/problems/critical-connections-in-a-network/, https://nthomas.org/2020-05-16-critical-connections-leetcode/

Hints

Hint 1: How can we best represent this problem in a data structure, and how may we traverse it efficiently to check the critical paths condition? (graph with adjacency list and dfs)

Hint 2: What is the relationship between the critical path condition and concepts like cycles or connected components? How can this problem be redescribed using those terms, and can we do any precomputation to allow us to more efficiently check for the critical path condition? (answer: yes, compute strongly connected components! See solution for why)

Solution

The key insight here is that we're looking for the bridge between strongly connected components. Luckily this is a solved problem, and Tarjan's algorithm will come to the rescue. Let's come up with a quick definition of a strongly connected component that works for us. A strongly connected component can be thought of as a graph in which no one edge is the weak link in the graph. Or, all edges have at least one backup to keep the graph tied together if it disappears.

If we take a set of nodes and number them in the order we visit them (like a timestamp, or just a monotonically increasing id), we can end up building an array of IDs. Let's call that the dfsNumber array, is for any node in the order we visited it in a depth first search.

Additionally, we want to know how far "backwards" into a graph a node can reach. If we know nothing about a node, the "oldest" seen node that node N can see is itself. Once we start to look at neighbor nodes, we can assert that the oldest node a neighbor can reach is the oldest node any of its neighbors can reach (ignoring the direct parent we came from). We can hold oldestReachable in an array, where oldestReachable[i] indicates the oldest timestamp reachable from node i:

```
int timestamp = 0;

void tarjan(int node, int &parent, vector<vector<int>> &adjacency, vector<int>
&dfsNumber, vector<int> &dfsLow, vector<pair<int,int>> &criticalEdges) {
    timestamp++;
    dfsNumber[node] = timestamp;
    dfsLow[node] = timestamp;

    for (auto neighbor : adjacency[node]) {
      if (neighbor === parent) continue;
      if (dfsNumber[neighbor] == 0) tarjan(neighbor, node, adjacency, dfsNumber,
dfsLow, criticalEdges);

      // resetting oldestReachable if neighbor can cycle back
      dfsLow[node] = min(dfsLow[node], dfsLow[neighbor]);

      // if neighbor cannot reach back to me or oldest, we have a critical edge
      if (dfsLow[neighbor] > dfsNumber[node]) edges.push_back({node, neighbor});
    }
  }


vector<pair<int,int>> &criticalConnections(vector<pair<int,int>> connections) {
  int n = connections.size();
  vector<vector<int>> adjacency(n, vector<int>());
  for (auto &p : connections) {
    adjacency[p.first].push_back(p.second);
    adjacency[p.second].push_back(p.first);
  }

  vector<int> dfsNumber(n,0);
  // this is what we called oldestReachable
  // the lowest number reachable by dfs from a node
  vector<int> dfsLow(n,0);
  vector<pair<int,int>> criticalEdges;
```

```
  tarjan(0, -1, adjacency, dfsNumber, dfsLow, criticalEdges);

  return criticalEdges;
}
```

Analysis

Time: O(|V|+|E|)

Space: O(|E|)

Tips and tricks

- Often with graphs it is useful to consider the relationship between the current problem and famous algorithms like Djikstra, Bellman-Ford, Floyd-Warshall, Ford-Fulkerson, so it may be useful to be familiar with such algorithms, especially when harder interviews are anticipated.
- A useful strategy may be to ask the question — how can we reformulate this problem in graph theory terms, e.g. using cycles, components, colorings, flow, etc? This generalizes to graph problems in general.

Follow-ups

- How might we do this if the critical elements were vertices and not edges? (e.g. buildings were blocked off, instead of roads) Solution: see https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/
- Can we solve this without Tarjan's algorithm, or view the problem in an alternative way? Possible solution: see https://leetcode.com/problems/critical-connections-in-a-network/discuss/382638/DFS-detailed-explanation-O(orEor)-solution