Additional Graph Algorithms

Oliver-Matis Lill

May 23, 2017
Today we will cover the following graph algorithms and data structures:

1. Disjoint-set forest
2. Floyd-Warshall all-pairs shortest path algorithm
Suppose we have an undirected (probably unconnected) graph and we have to perform the following queries on it:

1. Add an edge between vertices $u$ and $v$
2. Output whether there is a path from $u$ to $v$

The second query basically asks if $u$ and $v$ are in the same connected component.

The graph might look like this:
Disjoint-set Forest

Idea

- For each connected component pick a "master" node that stores all information about that component.
- Each node that’s not a master of a component will have an immediate master node.
- Traversing masters will lead to the master of the whole connected component.
- When two components are connected by an edge, make the master node of the first component the immediate master of the master node of the second component.
- To check if two nodes are in the same component, check if they have the same topmost master.
Disjoint-set Forest

Example

- The graph might look like this:

- And its Disjoint-set forest could look like this:
Disjoint-set Forest Problem

Suppose we have the following graph:

With the following Disjoint-set forest:

What is the worst case performance of finding the topmost master for a node?
Can we improve that?
When uniting two components, pick the new master from the larger component:

As a result, when a node gets a new topmost master, the new topmost master will cover at least 2x as many nodes.

Since a component can have at most $n$ nodes, the topmost master for a node can change at most $O(\log n)$ times.

As a result, the distance to the topmost master can also be at most $O(\log n)$. 

---

Disjoint-set Forest Fix

- When uniting two components, pick the new master from the larger component:

- As a result, when a node gets a new topmost master, the new topmost master will cover at least 2x as many nodes.

- Since a component can have at most $n$ nodes, the topmost master for a node can change at most $O(\log n)$ times.

- As a result, the distance to the topmost master can also be at most $O(\log n)$.
Example

```cpp
bool unite(Node* u, Node* v) {
    u = u->getMaster();
    v = v->getMaster();

    if(u == v)
        return false;
    if(u->size < v->size)
        swap(u, v);
    v->master = u;
    u->size += v->size;
    return true;
}
```
Suppose we have a weighted dense graph (for example $|E| \in \Theta(|V|^2)$) and we want to find a shortest path between each pair of vertices.

What previously covered algorithm could we use here?
Example

```c
void floydWarshall() {
    for(int k=0;k<n;k++)
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

- Basically it goes through each "mid-node" \( k \)
- For each \( k \) it goes through every pair of nodes \( (i, j) \) and adds a new "compound" edge between them corresponding to the traversal of existing (possibly compound) edges \( (i, k) \) and \( (k, j) \)
Proof

- We’ll show that for any pair \((u, v)\), the algorithm will ultimately create a compound edge between them corresponding to the shortest path from \(u\) to \(v\).
- Suppose that the shortest path from \(u\) to \(v\) is \([p_1, p_2, \ldots, p_l]\) where \(p_1 = u\) and \(p_l = v\).
- If the first "mid-node" in the path the algorithm reaches is \(p_i\), then it will create a compound edge between \(p_{i-1}\) and \(p_{i+1}\) corresponding to the path \([p_{i-1}, p_i, p_{i+1}]\).
- As a result after this there will be a new, as short, shortest path \([p_1, p_2, \ldots, p_{i-1}, p_{i+1}, \ldots, p_l]\) consisting of \(l - 1\) nodes.
- By applying the above repeatedly to the current shortest path, we see that we will eventually have createad a shortest path \([p_1, p_l]\) consisting of the desired single compound edge.
The shortest path progression could look like the following:

1 → 2 → 3 → 4 → 5
1 → 2 → 4 → 5
1 → 4 → 5
1 → 5