CS 594: Representations in Algorithm Design **Spring 2022**

Lecture 2: 01/13/2022

Lecturer: Xiaorui Sun *Scribe: Nima Shahbazi*

1 Last Lecture's Review

In the last lecture, we reviewed several representations of the graph. Graph is a very powerful representation by itself, however, we want to simplify a graph to make it have better properties so that more information can be computed from the abstract graph.

- **(Spanning) Tree:** Given a graph, we only keep a subset of edges and throw away the rest such that the remaining graph is a tree (no cycle) and it preserves the connectivity of the entire graph.
- **DFS and BFS Tree:** Special spanning trees with additional properties. BFS preserves the property of distance. DFS makes sure all the non-tree edges connected to the vertices such that one is the other's ancestor. Using these trees, based on different properties, we can solve different interesting problems, for example using DFS tree we can identify bridges of the graph while BFS trees can be used to compute distance.
- **Claim:** For a graph, with minimum $deg \geq 3$, there is a cycle of the graph of size $\leq O(\log n)$.

  (If the minimum $deg = 2$, the graph can be a single cycle of length n, but if the minimum deg increases by 1, there must exist some smallest cycle of size $\leq O(\log n)$ in the graph.)

Relevant Readings:

- Lecture 1 scribe notes
- Review on graph and tree basics and properties (from CS401)
- ...

2 Proof of The Claim

Claim: For a graph, with minimum $deg \geq 3$, there is a cycle of the graph of size $\leq O(\log n)$.

Proof: Starting with a BFS tree, we use two properties of BFS trees:

1. Edges connect vertices at the same or adjacent level.
2. If $\exists$ a non-tree edge that has one end-point on level $i \implies$ cycle of size $\leq 2i + 2$

First Property Proof: Assume Figure 1 is a BFS tree, and the red edge is a non-tree edge with one endpoint at level i, then the other endpoint must be at level $i-1$ or level i or level $i+1$, because if it is at a level smaller than $i-1$ then the green vertex should be at a level smaller than i (the distance from the starting vertex to this vertex will be smaller than i so this vertex will be at level smaller i). Similarly, if it is at a level greater than i, the other endpoint must be at most at level $i+1$ because this edge can be used as a tree edge to construct BFS tree.

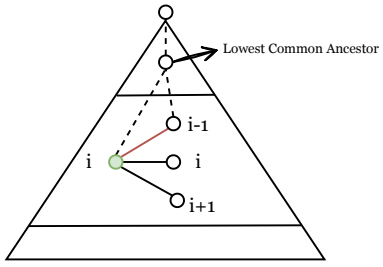

Figure 1: Example of a BFS tree

Second Property Proof: When you look at the paths in the BFS tree from the two endpoints of the red edge in Figure 1 to the root, these two paths must meet somewhere in the middle, because both paths at the end will go to the root so they must share some vertices or edges at some point. The first shared vertex is called *lowest common ancestor*. If you look at the path from one endpoint to the lowest common ancestor and the other endpoint to the lowest common ancestor, together with the non-tree edge (red edge) form a cycle of length $2i + 2$ (because one path to the root has at most i edges and the other path to the root has at most $i+1$ edges and together with the non-tree edge itself the cycle size would be: $i + i + 1 + 1 = 2i + 2$)

Second property shows that if we want to prove the claim, we only need to show that in the BFS tree, there must be some non-tree edge with one endpoint at level $\leq O(\log n)$.

Now let us have a closer look at the BFS tree based on the condition that for all vertices in the graph, $deg \geq 3$ in Figure 2. Starting with and arbitrary vertex v, it has all of its neighbors at level 1 of the BFS tree. If the vertices at level 1 have a non-tree edge in the BFS tree, then we are already done (we'll have a cycle of size at most 4). For each vertex at level 1, there is an edge to its parent vertex v and every other edge goes to the next level and in particular all of these edges should also be tree-edges cause otherwise we again have a non-tree edge and then we are done and so on so forth. So in general, for some integer i, if there does not exist a non-tree edge at level i or smaller, then each vertex will have a tree-edge in level $i-1$ and all the other edges must be tree edges and go to level $i+1$ and if at some level i there is a non-tree edge, the cycle will be of size $2i+2$, so this property holds all the way to $\log n$ level. Now, we use this property to see the number of vertices at each level (shown in Figure 2. So all the argument to this moment is that we don't have a cross edge at the level $\log n$ using that property. However, there's a contradiction because the number of vertices in total before the level $\leq \log n = 1 + 2 + 4 + ... + n = 2n - 1$, while we only have n vertices in the graph. Therefore, our initial assumption of graph not having any cross (non-tree) edge before level $\log n$ is not right and there must be some non-tree edge in the BFS tree prior to level $\log n$, so, we have a cycle of size $O(\log n)$.

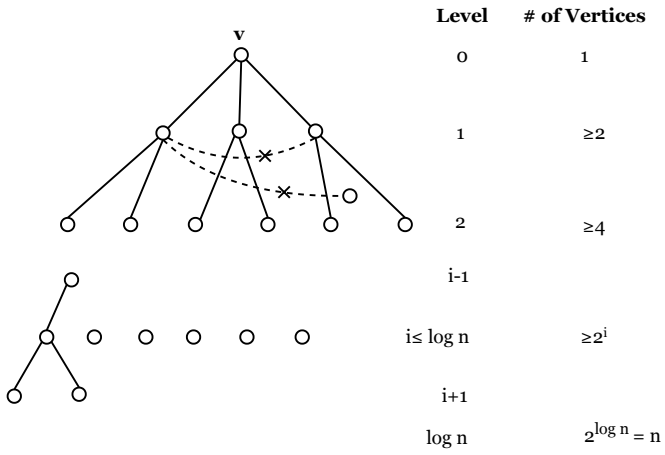

Figure 2: Number of vertices in each level of the BFS tree

This is an example to suggest that in many graph problems, tree is a basic structure

and provides a lot of nice properties. Some properties that at first glance may not relate to tree can be proven by a tree, so, tree representation if a graph is always the first step to solve a problem. Sometimes it is sufficient and sometimes the computation of the simpler version of the problem gives you a good approximation of the solution.

3 Dynamic Algorithms for Graph Connectivity

3.1 Motivation

In the modern big data era, the input to many problems are super large which causes a lot of problems like the cost of computation and running time. Additionally, in the big data era the input size is not static anymore and keeps changes for example in social networks, a person becomes friend with a new person or maybe unfriend each other but the point is that a super large graph may change overtime with respect to number of edges, vertices, connected components, diameter of the graph, etc. This change of graph, requires redoing the computation which is super expensive due to the size of the graph. Therefore, people started to consider that if they have the solution before the change in graph, is it possible to get the solution after the change in a more efficient manner? This question makes sense, especially because in practice for large inputs the change is frequent but is also small. For example each change in a social network graph is minor compared to the size of the graph. So if we are able to incorporate these small changes into the super large input, then potentially we can avoid to do the re-computation from scratch. This is called dynamic algorithms. The idea of a dynamic algorithm is to maintain some data structure so that updates can be quickly incorporated. Usually, we are interested in the problems that the update can be handled in $o(n)$ time. Let us see this in an example of how it is not needed to redo the whole computation and it is possible to just keep track of the change:

Example: Sum problem
Input: n integers $a_1, a_2, ..., a_n$
Update: $a_1 \longrightarrow a_1'$
Goal: Maintain S as sum of a_i
Claim: $O(1)$ to maintain the sum
Proof: If $a_i \longrightarrow a_i'$ then $S \longrightarrow S + a_i' - a_i$

3.2 Dynamic Connectivity

Problem Definition:
Input: Undirected graph with n vertices
Update: Add or delete and edge
Query: For vertices x and y, is x connected to y or not?
Goal: Update in $o(n)$

In this setting, the algorithm can process the initial graph, but after the initial processing, there will be updates or queries (with no particular order of operation or knowing the type of update).

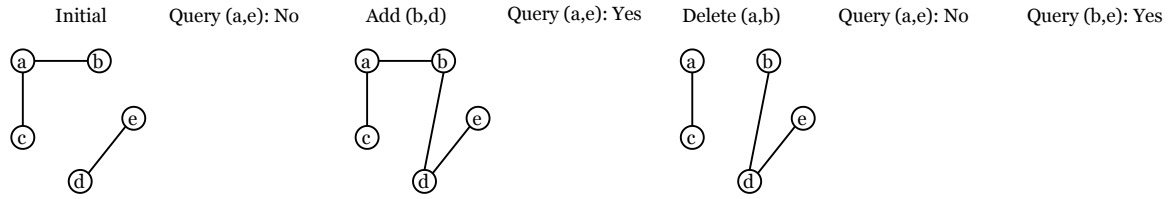

Figure 3: Example of dynamic connectivity

Theorem: Dynamic connectivity can be solved in $O(\log n)$ time with a randomized algorithm (best current deterministic algorithm time is $O(n^{o(1)})$).

In order to solve this problem, we need to think which kind of data structure we need to maintain, in order to update and query graph connectivity and which kind of information we need for the connectivity. Since we want to answer the query efficiently, we don't need to keep all the information of the graph. A spanning forest (spanning tree for each connected component) is the data structure needed to maintain the connectivity because it preserves the connectivity between any two vertices, so if we are able to maintain a spanning forest for the graph with respect to the updates of the graph, then this spanning forest is sufficient to tell us the connectivity:

Two vertices connected in spanning forest $F \iff$ Two vertices connected in G

So whenever given any two vertices, we only need to check if they belong to the same tree, if yes they are connected in the graph, otherwise no.

Goal: Maintain a spanning forest F of G in $O(\log n)$ time complexity.

Here we consider a few different scenarios for the updates with an example. In Figure 4, graph G is illustrated and the spanning forest is marked by the blue edges. G is not connected and there are two trees in this forest.

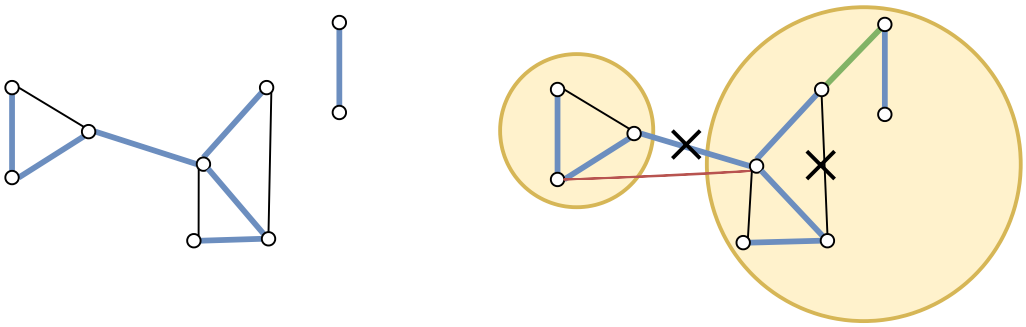

Figure 4: (Left) Graph G and spanning tree F. (Right) Four update scenarios in dynamic connectivity.

- Assume at the first step, the update is adding an edge (red edge). If the following edge is added to the graph, is there any change that need to be applied to the spanning forest? No and it is because this update inserts and edge between two vertices in the same connected component. So adding this edge does not connect the two trees together. So before and after the update, it is still a spanning forest and therefore no update is required.

- In the second step, we remove an edge from the graph. Is there any change that need to be applied to the spanning forest? No and it is because this update removes an edge that does not belong to the spanning forest.

- In the third step, we add an edge (green edge) to the graph. Is there any change that need to be applied to the spanning forest? Yes, because previously the graph had two connected components but with this edge, it becomes a single connected component. So to insert an edge connecting two different connected components, we need to add the newly added edge to the spanning forest F.

- In the last step, we remove a spanning tree edge from the forest and now F has two connected components, however, if we look at the entire graph, we still have a single connected component (using the red edge). So, in order to maintain the spanning forest property, we need to add the red edge which previously was a non-tree edge to the spanning tree. Therefore, if we delete a tree edge, we need to find a replacement if possible.

Given a spanning forest, distinguishing these four cases is easy, because what we need to do for the insert case is to check if two endpoints of an edge belongs to the

same connected component (done with a query) and for the delete case, we need to check if the edge is in the spanning forest or not. The first three cases are simple but how do we find a replacement for the fourth case? The naive way is to go over all the edges in the first connected component and check if there is an outgoing edge to the other connected component but potentially can be super slow.

Idea: "Sample" an edge going out of one connected component.

Let us consider a simple case illustrated in Figure 5:

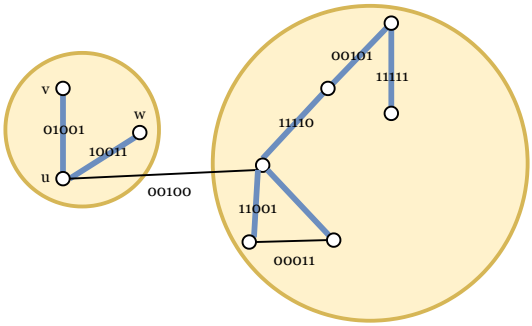

Figure 5: Recovering the edge connecting the two connected components

Idea: Edge going out of the connected component is unique. Assign a random binary name of the same length to each edge.
S_v is the binary sum (XOR) of names of incident edges to vertex v.
$S_v = 01001, \quad S_u = 01001 \oplus 10011 \oplus 00100 = 11110, \quad S_w = 10011$
$S_{cc} = \oplus_{v \in cc} = 01001 \oplus 11110 \oplus 10011 = 00100$

S_{cc} is exactly the name of the edge connecting two components. It is because if an edge has two endpoints in the same connected component they will cancel each other out in the sum and only the edge going out of the component will remain. Therefore we can recover the edge connecting two components using the sum of the edges in the connected component.

But what if multiple edges go out from one connected component to the other connected component? We will explain that in the next lecture.