15-492: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Graphs I **Date:** October 2nd
Scribe: Kanat Tangwongsan (based on Guy Blelloch's old notes)

11.1 Announcements

- **Reminder.** Assignment #3 is due Thursday October 4, 2007. There are some non-trivial parts.
- The midterm exam will be a take-home exam. We plan to hand it out on Monday October 15. It will be due back the next day—you have 24 hours to complete it. Details will follow soon.
- **Guest Lecture.** Vijay Saraswat (IBM Research) will be giving a guest lecture on Thursday October 11. Personal website: `http://www.saraswat.org/`

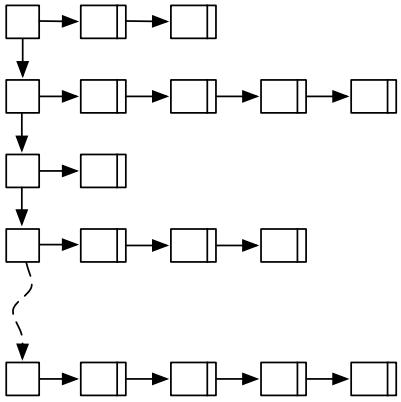
11.2 Today

- Graph Representations
- Connectivity and Minimum Spanning Tree.

11.3 Representing Graphs

We will look at different ways of representing graphs: adjacency lists, adjacency arrays, edge arrays, matrix, etc. Among these we probably are already familiar with adjacency lists and adjacency matrices.

– An adjacency array is simply representing the entries of an adjacency list in a (nested) array. The figure below shows an adjacency list.



The adjacency array representing the same graph looks something like `[[1, 2], [2, 4, 5, 6], [1], [3, 5, 7], ..., [4, 1, 3, 5]]`.

– An edge array (or list) is an array (or a list) that contains all the edges in the graph.

11.4 Graph Connectivity

Consider the following problem:

> **Problem:** Find all connected components in an *undirected* graph. That is, label all nodes in the same component with the same label.

Let m be the number of edges and n be the number of nodes in a graph. If we represent graphs with adjacency lists, depth-first or breadth-first search is $O(m)$, since each edge gets visited exactly once. But these searches are difficult to parallelize, because depth-first search is inherently serial and we cannot parallelize breadth-first search beyond the diameter of the graph (consider the pathological case of a linked list). So we resort to graph contraction to achieve better parallelism. At a high level, here is how the algorithm works:

1. Identify a set of connected clusters.
2. Contract each cluster into a node.
3. Repeat until one node remains.

11.4.1 Random mate graph contraction

Given a graph G (e.g. Figure 11.4.1), every node randomly flip an unbiased coin to select nodes as parents and children (see Figure 11.4.2). Every child that neighbors at least one parent picks a parent to contract into. This can be implemented easily if we can concurrent write. All edges from the child are carried over to the parent. See Figure 11.4.3 for an example. Notice that one parent may have several children, but a child can have only one parent. On average, after each contraction at least 1/4 of the vertices will be removed. The graph after contraction is shown in Figure 11.4.4. We continue this process until eventually only one node is left. If we need to label each component, we have to expand the graph after all the contractions are finished.

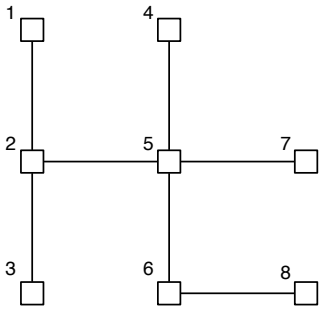


Figure 11.4.1: Input graph $G = (V, E)$

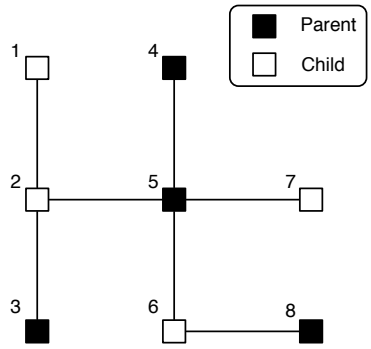


Figure 11.4.2: Selecting parents and children by coin flipping

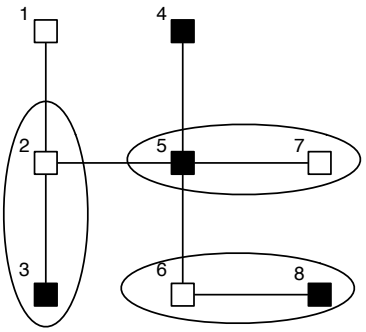


Figure 11.4.3: Contracting the children

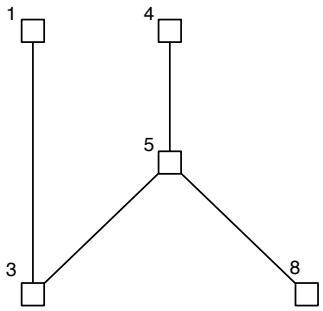


Figure 11.4.4: After contraction

Since we expect to remove $n/4$ nodes in each round, we finish in $O(\log n)$ rounds in expectation. This algorithm therefore has work $O(m \log n)$ and depth $(\log n)$. With some more work, we can also show high probability bounds for this algorithm. Note that this algorithm is *not* work efficient, because a straightforward DFS gives us an $O(m + n)$ algorithm.

At this point you may wonder if we need randomization to get an efficient parallel algorithm for this problem.

11.5 Deterministic Graph Contraction

Consider the following graph:

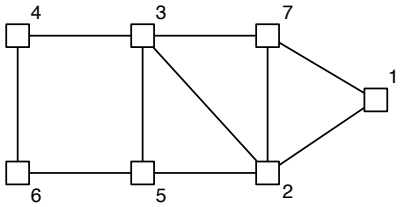


Figure 11.5.5: Sample graph

What we really want to do is contract all the nodes of the graph into a single node. If we had a spanning tree that was imposed on top of the graph, we could contract the graph by contracting the tree. The requirement for this tree is that it spans the whole graph and does not include any cycles. Unfortunately this turns out to be as hard as finding the connected components of the graph. Instead we will settle for finding a number of trees that cover the graph, contract them and then deal with the remaining nodes in the same manner.

The basic idea is to hook the nodes into one or more trees, contract the trees and then repeat the process until the entire graph is contracted into a set of unconnected nodes. These will be the connected components.

One way to get a tree that doesn't have a cycle is to require that the node of a tree have a smaller index than any of the children below it. Each edge in our graph connects two nodes, each of which has a distinct index. Thus, one way to get such a tree from the graph is to have each edge write the smaller of the two indices that it connects into the node with the larger index.

A single node may be connected to multiple other nodes, all of which might attempt to write a value into the node. One of these writes will be the last and will thus succeed in connecting the two nodes into a tree. After this first step we might find ourselves with a graph that looks like this:

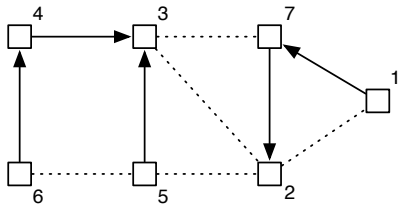


Figure 11.5.6: Graph after first hooking operation (solid edges are tree edges).

A False Claim. We remove half of the nodes in every contraction step. If every node were connected by a tree then we could remove half of the nodes with a single contraction of the tree. The problem is that we can't guarantee that all nodes will be connected—consider this counter example:

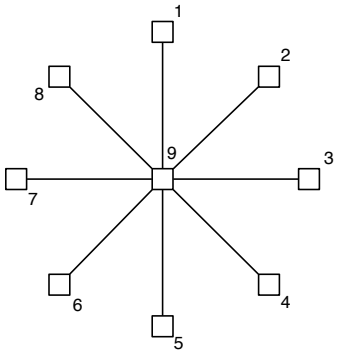


Figure 11.5.7: Counterexample

In the hooking phase, all edges will write the value of the outer nodes into the center node. Only one write will succeed and the resulting graph will connect the inner node to only one of the other nodes. Thus each hooking step removes just a single node from the graph.

One solution to this problem is to alternate hooking lesser nodes to greater nodes and then greater nodes to lesser nodes. This guarantees that every node will be hooked on one of these steps, and that we will end up removing at least half the nodes on each pair of steps (only the root of each tree needs to remain).

11.5.1 Graph contraction and pointer jumping

In the above algorithm we were required to contract the trees that we built. One way to do this is by *pointer jumping*. First make the root point to itself so that it is the only node that is its own parent.

Pointer jumping consists of taking each node in the graph and moving its pointer to instead point to its parent's parent. (Point yourself at your grandparent.) We can show that tree contraction is guaranteed to terminate in $O(\log d)$ steps where d is the depth of the tree. The way we determine whether we've contracted the entire tree is to check after two iterations of pointer jumping if everyone still points to the same place. If so then we're done.

Note that in practice the trees encountered are not very deep so pointer jumping works fine. In theory we should use a work-efficient version of tree contraction (discussed in lecture 13).

After we've contracted the tree we are still left with the nodes of the original graph that were not connected to the same tree. In the above example we would have node 1 and 2 left. We would then run the algorithm again on these remaining nodes.

If we use a work-efficient algorithm, contraction of the tree has work complexity $W(V, E) = O(|E|)$ and step complexity $S(V, E) = O(\log |V|)$. After each step of the algorithm (hooking and tree contraction) we are guaranteed to have eliminated at least half the nodes in the tree. Thus the work complexity of the entire algorithm is $W(V, E) = (|E| \log |V|)$ and the step complexity is $S(V, E) = O(\log^2 |V|)$.

In practice the time is much less than this upper bound. Only in pathological cases does the time really take $O(\log^2 |V|)$, and in these rare cases a renumbering of the nodes will usually reduce the time.