15-499: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Graphs I **Date:** February 24, 2009
Scribe: Jose H Sandoval

13.1 Graphs

13.1.1 Representation

A common way to represent the adjacency of nodes in a graph would be with an adjacency list. In this approach, we create a list for every node in the graph and keep adding its neighbors to the end of the list.

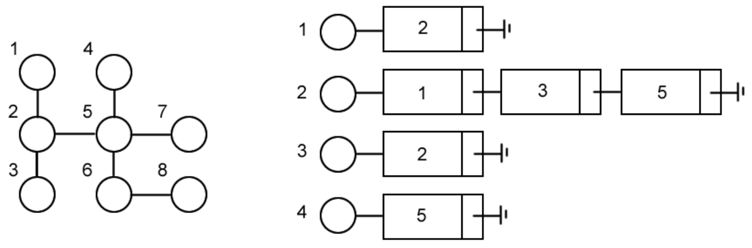

Figure 13.1.1: A graph and its correspondent adjacency list

Nevertheless, since using a list, this approach does not provide an easy way to run calculations in parallel. Different representations, such as adjacency matrices, binary trees or edge arrays, can give us better results.

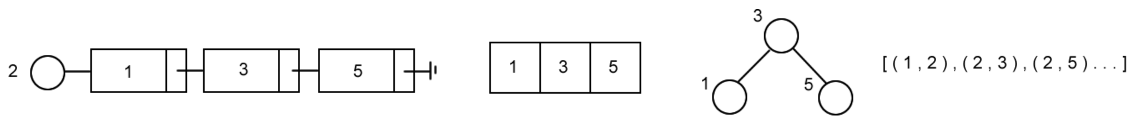

Figure 13.1.2: A graph can also be represented by an adjacency matrix, a binary tree or an edge array

13.1.2 Graph connectivity

Problem: Find all connected components in a graph [label every vertex with a representation of its component], [spanning tree].

Sequential solution: Depth first search.

1. Solve for a smaller subset.
2. Join adjacent nodes.
3. Label nodes with ones.

4. Repeat contraction until single vertices.

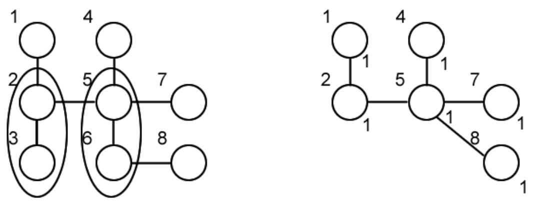

Figure 13.1.3: Contraction of a graph after one iteration

We can use random mate to decide what vertices to contract at every step. We then flip a coin on every vertex and contract only those that obtained heads and are pointing to a vertex that obtained tails. A problematic case will arise when two vertices that obtained heads attempt to contract the same vertex at the same time, creating a race condition. Nevertheless, as this race condition does not affect the result in a significant way, we can allow this to happen. This will result in the node that obtained tails being contracted to only one of the nodes that obtained heads.

Using random mate

1. Flip coins on every vertex.
2. Contract only heads that are pointing to tails.

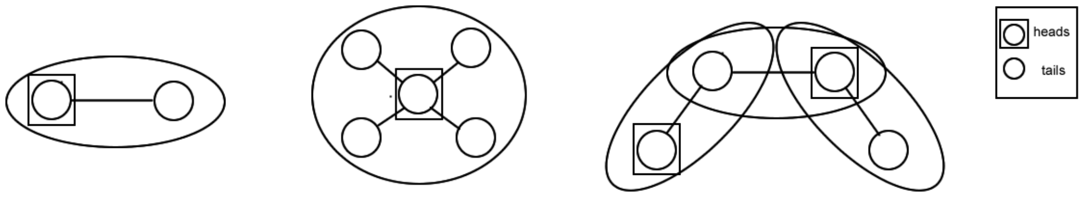

Figure 13.1.4: Different situations of graph contraction. The rightmost graph creates a race condition.

```
repeat
   Flip a coin on every vertex
   for all edge (u, v) do
      if u is a head and v a tail then
         Relabel v with u
         L[v]=u
      end if
   end for
   for all edge (u, v) do
      (L[u], L[v])
   end for
```

```
    Remove self edges
    Until edge array is empty
until edge array is empty
```

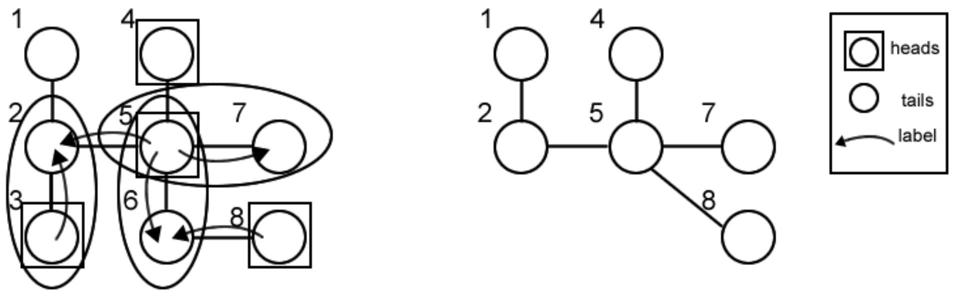

Figure 13.1.5: Graph contraction with race conditions.

Figure 13.1.5 shows two cases of race condition. Vertex 5 and vertex 3 will attempt to contract vertex 2 at the same time, while vertex 5 and vertex 8 will try to do the same to vertex 6. After concurrent labeling is applied by all vertices, the result is the graph shown on the rightmost figure. The corresponding label array and edge array look as follows:

$$L = [1, 2, 3, 4, 5, 6, 7, 8] \rightarrow [1, 3, 3, 4, 5, 5, 5, 8]$$
$$E = [(2 , 3) \dots] \rightarrow [(3, 3) \dots]$$

This algorithm consists in O(log n) rounds, each one of them O(log n). The probability of contraction in this case is greater than $\frac{1}{4}$, as one vertex that obtained heads could be connected to more than one vertex. If the graph starts with n^2 edges, the algorithm reduces n after its first iteration. This algorithm is not quite work efficient.

$$Depth = O(log^2n)$$

$$Work = O(mlogn)$$

13.2 Minimum spanning trees

Priority write: Writes with a priority and that with the highest priority wins.

Examples of priorities: Smaller index wins, bigger index wins, etc.

$$\text{Graph} = (V, E)$$
$$\text{Edges from U to } V - U$$
$$E' = \{ u, v \in E \mid u \in U, v \in V - U \}$$

Minimum spanning tree: Sum of its weights is minimum.

e = minimum edge from E', e in MST.