Recitation 8 — Graph Contraction

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

March 28, 2012

Today's Agenda:

- Connectivity
- Parallel MST

1 Connectivity

We're going to go over some examples from lecture in more detail today.

First, we're going to go over the LabelComponents problem: given a graph G, label each vertex so that two vertices have the same label iff they are in the same component (i.e. there is a path between them).

1.1 DFS

One sequential way to do this is with DFS.

```
structure DFSLabel : LABEL_COMPONENTS =
struct
  structure STSeq = STArraySequence
  structure Seq = STSeq.Seq
  open Seq

  type vertex = int
  type edge = vertex * vertex

  fun label (E, n) : labeling =
      let
        fun dfs L v label =
            case nth L v
              of SOME _ => L
               | NONE =>
               let val L' = STSeq.update (v, label) L
               in iter (fn (nbr,L'')  => dfs L'' nbr label) L' (neighbors E x)
               end

        val L = STSeq.fromSeq (tabulate (fn _ => NONE) n)
        val labels = iter (fn (v,L') => dfs L' v v) L (tabulate (fn v => v) n)
      in
        STSeq.toSeq labels
      end
end
```

What is the work/span of the above code? ($O(|E|+|V|)$ work and span)

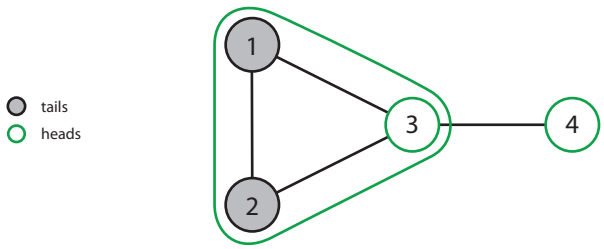
1.2 Union Find

Another way is with union-find. This is slower than DFS but easier to parallelize:

```
structure UFLabel : LABEL_COMPONENTS =
struct
  structure Seq = ArraySequence
  open Seq

  type vertex = int
  type edge = vertex * vertex

  fun label (E, n) : labeling =
      let
        fun contract ((x, y), L) =
            let
              val (x', y') = (nth L x, nth L y)
              val L' = update (x', y') L
            in
              map (nth L') L' (* <- What does this do?? *)
            end

        val L = tabulate (fn _ => NONE) n
      in
        iter contract L E
      end
end
```

What is the work/span of the above code? ($O(E^2)$ work, $O(|E|)$ span)

What does the highlighted line do? Why do we need it? This is a technique called path compression, which we can demonstrate with the following example on 4 vertices:



Observe that if we start with `L=⟨1,2,3,4⟩`, after the first round of contraction we get `L'=⟨3,3,3,4⟩`. That gives us the following contracted graph:

After contracting, we get `L'`=⟨3,3,4,4⟩. Notice that vertices 1 and 2 are still pointing to 3, but 3 is now part of component 4. Using `map (nth L') L'`, we *contract* each path of length 2 to length 1 and obtain `L''`=⟨4,4,4,4⟩.

The union-find algorithm above works by contracting each edge. It is slow because it only contracts one edge at a time. Maybe we can do better with star-contraction?

1.3 Star Contraction

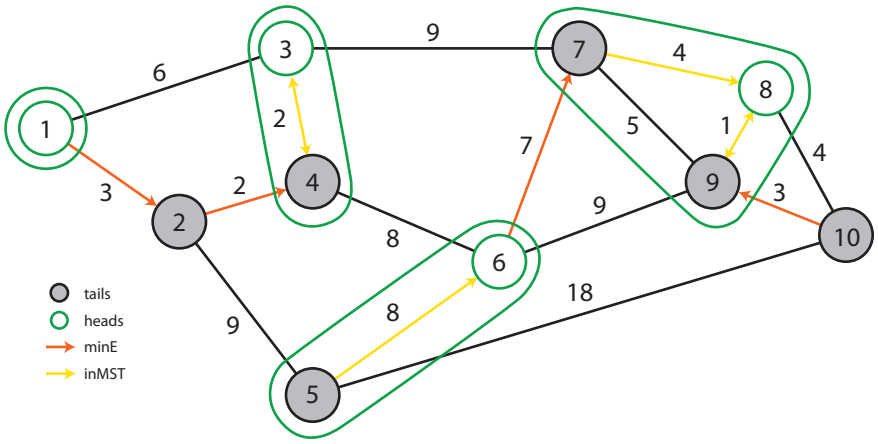
As in lecture, we flip a coin for each vertex, deciding if it will be the center of a star or a satellite. Then we associate each satellite with a center, and contract all of those edges. Now we can contract those edges, remove new self-loops, and recurse. Except for selecting the stars to contract, the code is very similar to union-find above:

```
structure StarContractLabel : LABEL_COMPONENTS =
struct
  structure Seq = ArraySequence
  structure Rand = Random210
  open Seq

  type vertex = int
  type edge = vertex * vertex

  fun label (E, n) =
      let
        fun LC (L : vertex seq, E, edge seq, seed : Rand.rand) =
            if length E = 0 then L
            else let
              val F = Rand.flip seed n
              fun isHook (u,v) = nth F u = 0 andalso nth F v = 1
              val hooks = filter isHook E
              val L' = inject hooks L
              val L'' = map (nth L') L'
              val E' = map (fn (u, v) => (nth L' u, nth L' v)) E
              val E'' = filter (fn (u, v) => u <> v) E'
            in
              LC (L'', E''. Rand.next seed)
            end

        val L = tabulate (fn i => i) n
      in
        LC (L, E, Rand.fromInt 0)
      end
end
```

2 MST

HW 7 will ask you to write parallel code to find an MST using the algorithm we presented in lecture 19. The idea is simple; instead of contracting any of the edges, only contract edges which are minimum weight from each vertex. Why does this work? Recall the Light Edge Rule / Cut Property from lecture.

Let's go over an example:



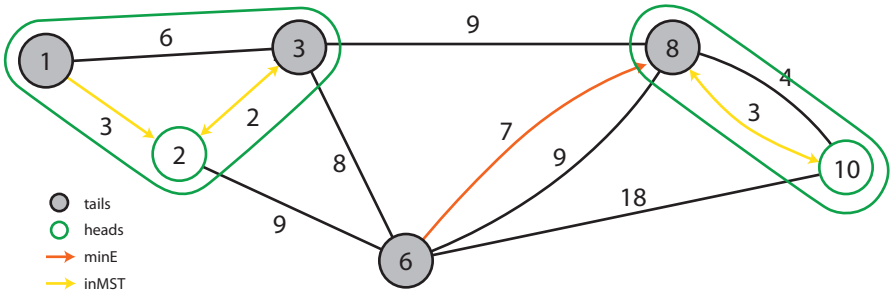
In the first round of the algorithm, we have the following flips:

1	2	3	4	5	6	7	8	9	10
H	T	H	T	T	H	T	H	T	T

Notice that vertices 3 and 4 are contracted, but 1 and 2 are not. Why?

A key point to note here is that in our version of the algorithm, we only consider the minimum *out*-edges from every vertex. That is to say, even though the input graph is undirected, we only pick an edge to be in our MST if it goes from tails to heads. This works because we represent undirectedness by having an edge in both directions.

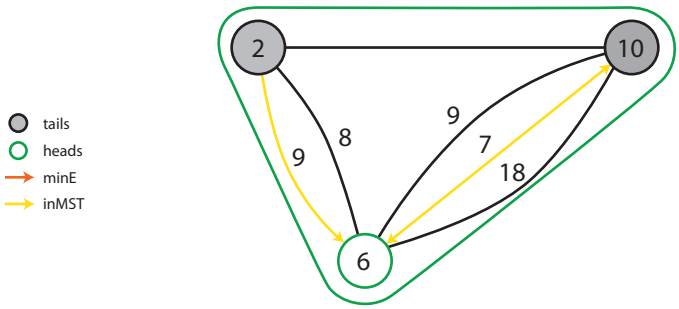
We get the following contracted graph in the next round:



Here is the sequence of flips generated for the second round:

1	2	3	4	5	6	7	8	9	10
T	H	T			T		T		H

We will generate another sequence of 10 flips here, but we only look at the ones generated for the vertices which remain in our contracted graph. This gives us:



with the following sequence of flips (ignoring vertices not in our graph):

1	2	3	4	5	6	7	8	9	10
	T				H				T

Of course, the flips seem a little fortuitous, allowing us to contract this graph in 3 rounds. That's because they were made up for this example. In general, it's not unreasonable to have a round of flips which results in no contractions at all. This is where expectation comes in.

Recall from lecture that each vertex has a minimum edge out which contracts with probability $1/4$. By Linearity of Expectation, $n/4$ vertices in expectation will be removed in each round.