Lecture 15 — Graph Contraction, Min Spanning Tree

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 18, 2011

Today:

- Analysis of Graph Contraction
- Minimum Spanning Tree

1 Graph Contraction on General Graphs

Last time, we described a form of graph contraction based on contracting stars. We begin this lecture by recalling the definition of stars.

Definition 1.1 (Star)**.** In an undirected graph $G = (V, E)$, a *star* is a subgraph of G with a center vertex v, a set of neighbors of $U_v \subseteq \{u \mid (v, u) \in E\}$ and the edges between them $E_{v,U} = \{(v, u) \mid u \in U_v\}$.

In fact, the star graph on $n + 1$ vertices is the bipartite graph $K_{1,n}$, which is a tree. The basic idea of star contraction is to identify and contract non-overlapping stars in a graph. For example, the following graph (left) contains 2 non-overlapping stars (right). The centers are colored red and the neighbors, green.

Based on the idea of stars, we can use the following algorithm for graph contraction:

```
1  % requires: an undirected graph G = (V, E) and random seed r
2  % returns: V' = remaining vertices after contraction,
3  %           P = mapping from removed vertices to V'
4  fun starContract(G = (V, E), r) =
5  let
6    % flip coin on each vertex
7    val C = {v ↦ coinFlip(v, r) : v ∈ V}
8    % select edges that go from a tail to a head
9    val TH = {(u, v) ∈ E | ¬C_u ∧ C_v}
10   % make mapping from tails to heads, removing duplicates
11   val P = ∪_{(u,v)∈TH} {u ↦ v}
12   % remove vertices that have been remapped
13   val V' = V \ domain(P)
14 in (V', P) end
```

The `starContract` procedure can be used for `identifyComponents` in the following generic graph-contraction algorithm:

```
1  fun graphContract((V, E), r) =
2  if |E| = 0 then V else
3    let
4      val (V', P) = identifyComponents((V, E), r)
5      val E' = {(P_u, P_v) : (u, v) ∈ E | P_u ≠ P_v}
6    in
7      graphContract((V', E'), next(r))
8    end
```

where $\overline{P_u}$ indicates the function that returns p if $(u \mapsto p) \in P$ and u otherwise (i.e., if the vertex has been relabeled it grabs the new label, otherwise it keeps the old one).

This returns one vertex for every connected component in the graph. We can therefore use it to count the number of components. To accomplish more interesting tasks, we have to augment the code similarly to we did with the generic BFS and DFS algorithms. Before looking at how to augment it, let's analyze the cost of the code. We say that a vertex is *attached* if it has at least one neighbor in the graph. We now prove the following lemma:

Lemma 1.2. *For a graph G with n attached vertices, let X_n be the random variable indicating the number of vertices removed by* `starContract`(G, r). *Then,* $\mathbf{E}[X_n] \geq n/4$.

Proof. Let H_v be the event that a vertex v comes up heads, T_v that it comes up tails, and R_v that it is removed. By definition, we know that an attached vertex v has at least one neighbor u. So, we have that $T_v \wedge H_u$ implies R_v since if v is a tail and u is a head v must either join $u's$ star or some other star. Therefore $\mathbf{Pr}[R_v] \geq \mathbf{Pr}[T_v]\,\mathbf{Pr}[H_u] = 1/4$. By the linearity of expectation, we have the number of removed vertices is

$$\mathbf{E}\left[\sum_{v:v \text{ attached}} \mathbb{I}\{R_v\}\right] = \sum_{v:v \text{ attached}} \mathbf{E}[\mathbb{I}\{R_v\}] \geq n/4$$

since we have n attached vertices. □

Exercise 1. *What is the probability that a vertex with degree d is removed.*

Now we can write a recurrence for the number of rounds required by the `graphContract` using `starContract` for each step. In particular, let n be the number of attached vertices and $R(n)$ be the number of rounds. When there are no attached vertices, the algorithm terminates since there are no edges and there cannot be one attached vertex, so we have $R(1) = 0$. Although vertices can become detached, they can never become attached again, so we have

$$R(n) \leq 1 + R(n - X_n).$$

Now recall that the Karp-Upfal-Wigderson lemma from last class gave us the solution

$$\mathbf{E}[R(n)] \;\leq\; \sum_{i=1}^{n} \frac{1}{\mu(i)}$$

where $\mathbf{E}[X_n] \geq \mu(n)$. This is the summation form; we also did an integral form. In this particular case, we have $\mu(n) = n/4$, giving a similar sum as we saw in the last lecture:

$$\mathbf{E}[R(n)] \leq \sum_{i=1}^{n} \frac{4}{i} = 4H_n = \Theta(\log n),$$

where H_n is the n-th Harmonic number. You should note that this sum will come up many times in the analysis of randomized algorithms.

As an aside, the Harmonic sum has the following nice property:

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n + \gamma + \varepsilon_n,$$

where γ is the Euler-Mascheroni constant, which is approximately $0.57721\cdots$, and $\varepsilon_n \sim \frac{1}{2n}$, which tends to 0 as n approaches ∞. This shows that the summation and integral of $1/i$ are almost identical (up to constants and a low-order term).

Using arrays, it is reasonably easy to implement each step of `starContract` using $O(n+m)$ work and $O(\log n)$ span. Therefore, since there are $O(\log n)$ rounds, the overall span of the algorithm is $O(\log^2 n)$.

Ideally, we would like to show that the overall work is linear since we might expect that the size is going down by a constant fraction on each step. However, this is not the case. Although we have shown that we can remove a constant fraction of the attached vertices on one star contract step, we have not shown anything about the number of edges. We can argue that the number of edges removed is at least equal to the number of vertices since removing a vertex also removes the edge that attaches it to its star's center. However, this does not help asymptotically bound the number of edges removed. Consider the following sequence of steps:

step	vertices	edges
1	n	m
2	$n/2$	$m - n/2$
3	$n/4$	$m - 3n/4$
4	$n/8$	$m - 7n/8$

In this example, it is clear that the number of edges does not drop below $m - n$, so if there are $m > 2n$ edges to start with, the overall work will be $O(m \log n)$. Indeed, this is the best bound we can show. All together this gives us the following theorem:

Theorem 1.3. *For a graph $G = (V, E)$, graph contraction using `starContract` with an array sequence works in $O(|E| \log |V|)$ work and $O(\log^2 |V|)$ span.*

2 Minimum Spanning Tree

The minimum (weight) spanning tree (MST) problem is given an connected undirected graph $G = (V, E)$, find a spanning tree of minimum weight (i.e. sum of the weights of the edges). You have seen Minimum Weigh Spanning Trees in both 15-122 and 15-251.

We believe in both classes, you went over Kruskal's algorithm. The basic structure was

```
sort edges by weight
put each vertex in its own component
for each edge e = (u,v) in order
   if u and v are in the same component skip
   else join the components for u and v and add e to the MST
```

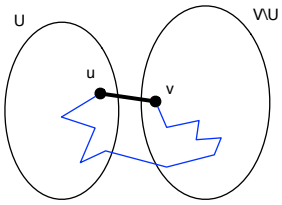
You used a union-find data structure to detect when two vertices are in the same component and join them if not.

The main property that makes Kruskal's algorithm as well as most other algorithms work is a simple fact about cuts in a graph. Here we will assume without any loss of generality that all edges have distinct weights. This is easy to do since we can break ties in a consistent way. For a graph $G = (V, E)$, a *cut* is defined in terms of a subset $U \subsetneq V$. This set U partitions the graph into $(U, V \setminus U)$, and we refer to the edges between the two parts as the cut edges $E(U, \overline{U})$, where as is typical in literature, we write $\overline{U} = V \setminus U$. The subset U might include a single vertex v, in which case the cut edges would be all edges incident on v. But the subset U must be a proper subset of V (i.e., $U \neq \emptyset$ and $U \neq V$).

The property that we will use is the following:

Theorem 2.1. *Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights. For any nonempty $U \subsetneq V$, the minimum weight edge e between U and $V \setminus U$ is in the minimum spanning tree MST(G) of G.*

Proof. The proof is by contradiction. Assume the minimum-weighted edge $e = (u, v)$ is not in the MST. Since the MST spans the graph, there must be some simple path P connecting u and v in the MST (i.e., consisting of just edges in the MST). The path must cross the cut between U and $V \setminus U$ at least once since u and v are on opposite sides. By attaching P to e, we form a cycle (recall that by assumption $e \notin MST$). If we remove the maximum weight edge from P and replace it with e we will still have a spanning tree, but it will be have less weight. This is a contradiction. □



Note that the last step in the proof uses the facts that (1) adding an edge to a spanning tree creates a cycle, and (2) removing any edge from this cycle creates a tree again.

This property was used in Kruskal's algorithm although we won't review this use here. Another algorithm that uses this property is Prim's algorithm. It basically uses a priority first search to grow the tree starting at an arbitrary source vertex. The algorithm maintains a visited set U, which also corresponds to the set U in the cut. At each step, it selects the minimum weigh edge $e = (u, v), u \in U, v \in V \setminus U$. This is in the MST by the theorem 2.1. It adds the adjoining vertex to U and e to the MST. After $|V|$ steps, it has added all vertices to the tree and it terminates. To select the minimum weight edge leaving U on each step, it stores all edges leaving U in a priority queue. The algorithm is almost identical to Dijkstra's algorithm but instead of storing distances in the priority queue, it stores edge weights.

Both Kruskal's and Prim's algorithms are sequential.

Here we show a parallel algorithm based on an approach by Borůvka that actually predates both Kruskal and Prim. We use graph contraction. The idea is actually quite simple. During graph contraction,

we can consider any of the contracted components as our set U. Therefore, the minimum edge out of each component must be in the MST. We can modify our `starContract` routine so that after flipping coins, the tails only attach across their minimum weight edge. The algorithm is as follows.

```
1  fun minStarContract(G = (V, E), r) =
2  let
3     val minE = minEdges (G)
4     val C = {v ↦ coinFlip(v, r) : v ∈ V}
5     val P = {(u ↦ (v, l)) ∈ minE | ¬C_u ∧ C_v}
6     val V' = V \ domain(P)
7  in (V', P) end
```

There is a little bit of trickiness since as the graph contracts the endpoints of each edge changes. Therefore if we want to return the edges of the minimum spanning tree, they might not correspond to the original endpoints. To deal with this we associate a unique label with every edge and return the tree as a set of labels (i.e. the labels of the edges in the spanning tree). We also associate the weight directly with the edge. The type of each edge is therefore $(\texttt{vertex} \times \texttt{vertex} \times \texttt{weight} \times \texttt{label})$ where the two vertex endpoints can change as the graph contracts but the weight and label stays fixed. The function `minEdges`(G) in line 3 finds the minimum edge out of each vertex v and maps v to the pair consisting of the neighbor along the edge and the edge label. Recall that by Theorem 2.1 since all these edges are minimum out of the vertex they are safe to add to the MST. Line 5 then picks from these edges the edges that go from a tail to a head, and therefore generates a mapping from tails to heads along minimum edges, creating stars. Finally line 6 removes all vertices that have been relabeled.

This can now be used in the following MST code, which is similar to the `graphContract` code earlier in the lecture except we return the set of labels for the MST edges instead of the remaining vertices.

```
1  fun MST((V, E), r) =
2  if |E| = 0 then {} else
3     let
4        val (V', PT) = identifyComponents((V, E), r)
5        val P = {u : (u, l) ∈ PT}
6        val T = {l : (u, l) ∈ PT}
7        val E' = {(P_u, P_v) : (u, v) ∈ E | P_u ≠ P_v}
8     in
9        MST((V', E'), next(r)) ∪ T
10    end
```