1 Graph Contraction

So far we have mostly talking about standard techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of these are easy to parallelize while others are not. For example, we saw there was plenty of parallelism in the Bellman-Ford algorithm, and also in the all pairs shortest path algorithms since they are based on parallel application of Dijkstra (and perhaps Bellman Ford preprocessing if there are negative weights).

We are now going to discuss some techniques that will add to your toolbox for parallel algorithms. The first of these techniques is graph contraction. This is actually a reasonably simple technique and can be applied to a variety of problems including graph connectivity, spanning trees, and minimum spanning trees. In the discussion of graph contraction, we will assume that the graph is undirected unless otherwise stated.

The basic outline of the approach is the following:

\[ \text{ContractGraph}(G = (V, E)) = \]

1. Identify a set of disjoint connected components in \( G \)
2. Pick a vertex to represent its components and contract other vertices in the same component into it.
3. Let \( V' \) be the set of vertices after contracting each component into its representing vertex.
4. Let \( E' \) be edges after relabeling each edge so its endpoints refer to the representing vertex.
5. Remove self-loops and parallel edges in \( E' \).
6. If \( |E'| > 0 \) then \( \text{ContractGraph}(G' = (V', E')) \)

We refer to each recursive call as a contraction step.

Now let’s go through some examples of how we might contract a graph. Consider the following graph:

\[ \begin{align*}
&\text{a} \quad \text{b} \\
&\text{c} \quad \text{d} \\
&\text{e} \quad \text{f}
\end{align*} \]
In this graph, we could identify the disjoint components \{a, b, c\}, \{d\}, \{e, f\}.

After contracting, we would be left with a triangle. Note that in the intermediate step, when we join \(a, b, c\), we create redundant edges to \(d\) (each one of them had an original edge to \(d\)). We therefore replace these with a single edge. However, in some algorithms, it is convenient to allow parallel (redundant) edges rather than going to the work of removing them. This is sometimes referred to as a multigraph.

Instead of contracting \{a, b, c\}, \{d\}, \{e, f\}, we could contract the components \{a, c\}, \{b, d\}, \{e, f\}. In this case, we would be left with three vertices connected in a line. In the two limits, we could contract nothing, or contract all vertices.

Why, you might ask, is contraction useful in parallel algorithms? Well, it is first worth noting that if the size of the graph reduces by a constant factor on each step, then the algorithm will finish after only \(O(\log n)\) steps. (This is the familiar recurrence \(f(n) = f(n/b) + c\).) Therefore, if we can run each step in parallel, we have a good parallel algorithm. However, even if we can contract in parallel, how can we use it to do anything useful? There are certain properties of the graph that can be maintained through contractions. For example, contraction maintains the connectivity structure of the graph. Therefore, if we start out with \(k\) connected components in a graph, we will end up with \(k\) components. So we can simply keep on contracting the graph until we run out of edges, then we check how many isolated vertices (i.e., vertices with not edges connecting to it) are left in the graph, and that tells us how many connected component there are in the original graph.

So, how do we select components? We must do so in parallel. Also ideally, the size of the vertex set after a contraction step should reduce by a constant fraction. Being able to do this efficiently and deterministically turns out to be quite a difficult problem.

2 Star Contraction and Connected Components

**Definition 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.
The vertex is called the star center. The neighbors can be connected with each other, but they are all viewed as contracting with \( v \).

How do we decide which vertex is the star center? We will use randomization. The basic approach is:

Find stars:
1. Every vertex flips an unbiased coin to decide if it a center
2. Every non-center tries to contract into (or hook up with) a neighbor that is a center
3. All the hooks identify a set of stars

It should be clear that stars are disjoint since every non center only hooks up with one center, and the center does not hook up with anyone. Also, one can break ties arbitrarily if there are multiple neighbors that can be a center.

How many vertices do we expect to remove?

\[
\text{STARCONTRACT}(V, E)
\]

1. **parallel_for** \( u \in V \)
   
   do flip a coin for each the vertex, \( C(u) \in \{H, T\} \)
2. For each vertex \( u \) with \( C(u) = T \), pick a neighbor \( v \) such that \( C(v) = H \) to contract into.
   If multiple such neighbors exist, arbitrarily pick one. If none exists, we will not contract \( u \) in this round.
3. Relabel vertices that have \( C(u) = T \) to the neighbor \( u \) choose to contract into. (Call this set of vertices \( M \).)
4. Remove vertices that got contracted, that is create \( V' \leftarrow V \setminus M \).
5. Create \( E' \): for every \( v \in V' \) and for every \( u \in M \) that contracted into \( v \), merge \( u \)'s adjacency list into \( v \)'s.
6. Relabel edges in \( E' \): for every vertex \( u \) contracted into \( v \), relabel \( u \) as \( v \) in the adjacency list.
7. Remove self loops in \( E' \).
8. **return** \((V', E')\)

The **STARCONTRACT** procedure can be used for **IDENTIFYCOMPONENTS** in the following generic graph-contraction algorithm:

\[
\text{IDENTIFYCOMPONENTS}(V, E)
\]

1. **if** \(|E| \neq 0\)
2. **do** \((V', E') \leftarrow \text{STARCONTRACT}(V, E)\)
3. **return** \( \text{IDENTIFYCOMPONENTS}(V', E')\)
4. **return** \(|V|\)
This returns one vertex for every connected component in the graph. We can therefore use it to count the number of connected components in the original graph.

First let’s think about how much time each call to STARCONTRACT takes. One reason we are not removing parallel edges in the relabeling step is because, it turns out that it’s difficult to do that in linear time. If the adjacency list for each vertex is sorted, we could potentially remove parallel edges in linear time during step 5 in STARCONTRACT. Unfortunately the list can become out of order after relabeling, so we say, let’s not. We keep the parallel edges; eventually when the two end points contract, they become self-loops, and we will remove them at that point. It turns out that, in either case the asymptotic bound is the same. Therefore, each round takes $O(m)$ time where we have $m$ edges.

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** 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, $E[X_n] \geq n/4$.

**Proof.** Let $H_v$ be the indicator random variable representing 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 $T_v \land H_u$ 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 $Pr\{R_v = 1\} \geq Pr\{T_v = 1\} Pr\{H_u = 1\} = 1/4$. By the linearity of expectation, we have the number of removed vertices is

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

since we have $n$ attached vertices.

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

If the number of vertices being removed at every round were exactly $n/4$, then what? What running time would we get? So what’s the overall expected work and span? 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 (and this is the case whether we remove parallel edges or not). 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, say we actually remove half of the vertices in each round:

<table>
<thead>
<tr>
<th>step</th>
<th>vertices</th>
<th>edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$n$</td>
<td>$m$</td>
</tr>
<tr>
<td>2</td>
<td>$n/2$</td>
<td>$m - n/2$</td>
</tr>
<tr>
<td>3</td>
<td>$n/4$</td>
<td>$m - 3n/4$</td>
</tr>
<tr>
<td>4</td>
<td>$n/8$</td>
<td>$m - 7n/8$</td>
</tr>
</tbody>
</table>
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, so the overall work in each round is still \( O(m) \). Therefore, in this informal analysis, the total work is \( O(m \log n) \) and the span is \( O(\log^2 n) \).

### 3 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). That is, find a spanning tree \( T \) that minimizes \( w(T) = \sum_{e \in E(T)} w_e \).

In previous classes you may have seem Kruskal’s algorithm:

1. sort edges by weight
2. put each vertex in its own component
3. for each edge \( e = (u, v) \) in order or weight
   - if \( u \) and \( v \) are in the same component skip
   - else join the components for \( u \) and \( v \) and add \( e \) to the MST
4. if (|\( T \)| = |\( V \)|) break;

We use a union-find data structure to detect when two vertices are in the same component and join them if not. With an efficient implementation of union-find, the work for the algorithm is dominated by the need to sort the edges. The algorithm therefore runs in \( O(|E| \log |V|) \) work. It is fully sequential.

You may have also seen Prim’s algorithm:

1. Start with an arbitrary source vertex in the graph, and maintain a set of visited vertices \( X \).
2. In each step, grow the tree maintained by \( X \):
   - add to \( X \) the vertex that is adjacent to the tree maintained by \( X \) that:
     1. has the smallest weight, and
     2. would maintain the tree property.
3. Repeat until all vertices have been added.

The Prim’s algorithm grows a tree (maintained by set \( X \)) starting from some arbitrarily chosen vertex, and only adds an edge \( e \) if \( e \) has the smallest weight leaving the tree maintained by \( X \) and if adding \( e \) does not break the tree property of set \( X \). To select the minimum weight edge leaving \( X \) on each step, it stores all edges leaving \( X \) in a priority queue and adds the edge only if the other end point is not in \( X \) already. The algorithm is almost identical to Dijkstra’s algorithm but instead of storing distances in the priority queue, it stores edge weights. Assuming you use a binary heap, it’s asymptotically the same as Kruskal’s (but you can do a little better if you use a Fibonacci heap), and it is also sequential.

At first glance, these two algorithms are quite different, but they both reply on the same underlying principles about “cuts” in a graph, which we’ll refer to as the **light edge property**. 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 \textbf{cut} is defined in terms of a subset \( U \subseteq V \). 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 \( U \neq V \)).

\textbf{Theorem 2 (The Light Edge Property)}  Let \( G = (V, E, w) \) be a connected undirected weighted graph with distinct edge weights. For any nonempty \( U \subseteq V \), the minimum weight edge \( e \) between \( U \) and \( V \setminus U \) is in the minimum spanning tree \( \text{MST}(G) \) of \( G \).

\textbf{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 \( e \notin \text{MST} \)). 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. \( \square \)

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.

If you think about it, both Kruskal’s and Prim’s algorithms work under this principle. For Kruskal’s, every edge that we add to the MST is a lightest weight edge remaining that would connect two distinct components. This edge must be in the MST by the Light Edge Property. For Prim’s, the algorithm maintains a visited set \( X \), which also corresponds to the set \( U \) in the cut. At each step, it selects the minimum weight edge \( e = (u, v) \), \( u \in X, v \in V \setminus X \). Again, this edge must be in the MST by the Light Edge Property.

There is yet another MST algorithm that you may not have heard of, called Borůvka, which actually predates both Kruskal’s and Prim’s (in 1926, before computers are even invented!). Again, it exploits the Light Edge Property, and the main observation is that, \textbf{the minimum weight edge out of every vertex of a weighted graph} \( G \) \textbf{belongs to its MST}. This observation follows directly from the Light Edge Property, if you consider each vertex as its own set \( U \).

The following example illustrates this situation, where we have highlighted the minimum weight edges. Note that some edges are minimum for both of their endpoints (i.e., the ones weighted 1 and 4).

Borůvka’s algorithm throws all the minimum weight edges coming out of each vertex into the MST, contracts these minimum weight edges, and repeats until no more edges left. Here, we show a parallel based on an approach similar to Borůvka’s algorithm.

We can modify the contract routine to only consider contracting the set of edges that are a subset of \( E \) that are the minimum edge weight coming out of each vertex, as follows:
**MinStarContract**(V, E, T)

1. Let \( \text{min}E \) be the set of all minimum-weight edge coming out of each attached vertex in \( G \).
2. Flip a coin to label each vertex, \( L(u) \in \{\text{head, tail}\} \).
3. Select vertex \( u \) to contract into \( v \) such that \( L(v) = \text{head} \) and \( L(u) = \text{tail} \), and \((u, v) \in \text{min}E\).
   Mark this edge \((u, v)\) as being part of MST \( T \).
4. Remove vertices that got contracted in step 3.
5. Create \( E' \) by relabeling \( E \) (i.e., rewrite vertex \( u \) that got contracted into \( v \) in step 3 as \( v \)).
6. Mark the contracted edges in \( T \) using its unique label as being in MST.
7. Relabel edges to create \( E' \): for every vertex \( u \) contracted into \( v \), change \( u \) to \( v \) in the adjacency list.
8. Remove self loops in \( E' \).
9. **return** \((V', E', T)\)

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. Also, we need to somehow avoid races on \( T \) in step 3. To deal with the first issue, we can simply associate a unique label with every edge. The end point of edges can change during contractions, but the edge labels will remain the same. To deal with the second issue, we simply store \( T \) as an edge array, mapping edge labels (the array index) to the edges in the original graph \( G \). To indicate that an edge should be included in \( T \), we simply look up in the array using the edge label and mark the edge as included.

Analysis of this algorithm is very similar to the SCC algorithm. Note that even though we only consider a subset of edges, the expected number of vertices removed is still \( n/4 \), because at each step, every attached vertex \( v \) has an edge in \( \text{min}E \), and the probability that \( v \) gets contracted is still \( 1/4 \).