

Lecture 18 — Graph Contraction II: Connectivity and MSTs

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

Lectured by Kanat Tangwongsan — March 22, 2012

The main theme of today's lecture is *graph contraction* and its applications to graph connectivity and minimum spanning trees.

1 Graph Contraction

Until recently, we have been talking mostly about techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of them are easy to parallelize while others are not. For example, we saw there is parallelism in BFS because each level can be explored in parallel, assuming the number of levels is not too large. But there was no parallelism in DFS. There was also no parallelism in the version of Dijkstra's algorithm we discussed, which used priority first search.¹ 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.

We are now going to discuss some techniques that will add to your toolbox for parallel algorithms. One technique that we saw last time was tree contraction, which is simple and extremely powerful when the structure we have is a tree. In this lecture, we will focus on *graph contraction*, a related technique for general graphs. This is a reasonably simple technique and can be applied to a variety of problems, including graph connectivity, spanning trees, and minimum spanning trees. Unlike in the tree setting, 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:

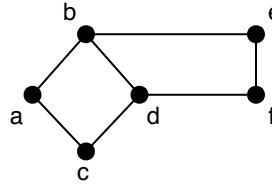
ContractGraph($G = (V, E)$) =

1. Identify a set of disjoint connected components in G
2. V' = the set of vertices after contracting each component into a single vertex
3. E' = after relabeling each edge so its endpoints refer to the new vertex
4. E'' = remove self-loops (and parallel edges)
5. If $|E''| > 0$ then **ContractGraph**($G' = (V', E'')$)

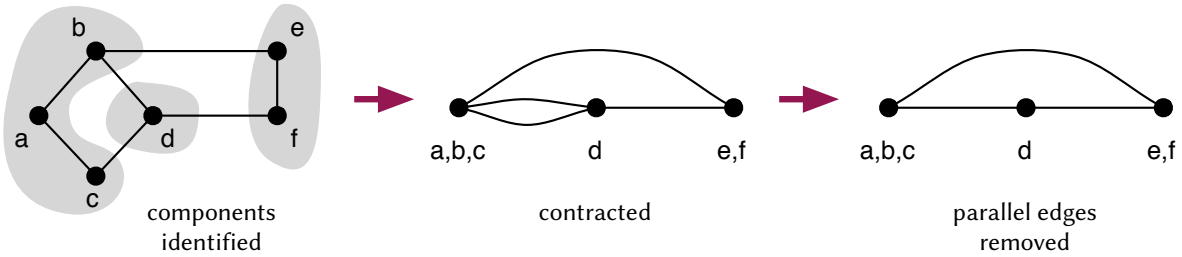
Each recursive call in this algorithm is called a *contraction step*. To begin, let's go through some examples of how we might contract a graph. Consider the following graph:

[†]Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

¹In reality, there is some parallelism in both DFS and Dijkstra when graphs are dense—in particular, although vertices need to be visited sequentially the edges can be processed in parallel. If we have time, we will get back to this when we cover priority queues in more detail.



Following the basic outline we just discussed, we first identify disjoint components—for example, in this graph, we might find disjoint components $\{a, b, c\}$, $\{d\}$, $\{e, f\}$, as illustrated in the figure below:



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, depending on the particular use, 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.

If instead we found the components $\{a, c\}$, $\{b, d\}$, $\{e, f\}$, then we would be left with three vertices connected in a line. Depending on how select the components, the number of vertices contracted can vary greatly: in the two limits, we could contract nothing or contract all vertices.

There are a couple special kinds of contraction that are worth mentioning:

Edge Contraction: Only pairs of vertices connected by an edge are contracted. One can think of the edges as pulling the two vertices together into one and then disappearing.

Star Contraction: One vertex of each component is identified as the center of the star and all other vertices are directly connected to it.

Tree Contraction: Generalizing star contraction, disjoint trees within the graph are identified and tree contraction is performed on these trees.

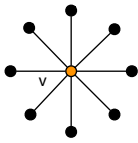
Why, you might ask, is contraction useful in parallel algorithms? First is the reason of efficiency. As we have seen in tree contraction, if the size of the graph reduces by a constant factor on each step, then the algorithm will finish after only $O(\log n)$ steps. This means that if we can run each step in parallel, we have a highly parallel algorithm. Such algorithms can be theoretically much more parallel than BFS since the parallelism is independent of the diameter of the graph or the shape of the graph. However, even if we can contract in parallel, how can we use it to do anything useful? One reason is that 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. It also turns

out that if we pick the edges to contract on carefully, then the contraction maintains other properties that are useful. For example, we will see how it can be used to find minimum spanning trees.

The next question to ask is: *how do we select components?* Keep in mind that we want to do this in parallel. On a cycle graph, we could use the familiar random pairing process adapted from the tree contraction algorithm to contract a constant fraction of the vertices each time in expectation. This is a form of edge contraction mentioned earlier.

Next we will look at star graphs. Here is a definition:

Definition 1.1 (Star). In an undirected graph $G = (V, E)$, a *star* is a subgraph H 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 words, a star graph is a graph made up of a single vertex v in the middle (called the center) and all the other vertices hanging off of it; these vertices are connected only to the center. Notice that a star graph is in fact a tree. If rooted at the center, it has depth 1, which is a super shallow tree.

If we're given a star, will edge contraction work well on this? How many edges can contract on each step. It is not difficult to convince ourselves that on a star graph with $n + 1$ nodes—1 center and n “satellites”—any edge contraction algorithm will take $\Omega(n)$ steps.

1.1 Star Contraction

Instead, we will consider more aggressive form of contraction that can contract disjoint stars in one step. The idea is to combine the star center with all its “satellites” all at once². To apply this form of contraction on a general graph, we have to be able to answer the question: *How can we find disjoint stars?* By disjoint, we mean that each node belong to at most one star. As an example, in the graph below (left), we can find 2 non-overlapping stars (right). The centers are colored red and the neighbors are green.



Finding Stars. One simple idea that has been fruitful so far is the use of randomization. Let's see what we can do with coin flips. At a high level, we'll use coin flips to first decide which vertices will be star centers and which ones will be satellites and after that, we'll decide how to pair up each satellite with a center.

²This is essentially running rake from tree contraction.

As usual, we'll start by flipping a coin for each vertex. If it comes up heads, that vertex is a star center. And if it comes up tails, then it will be a potential satellite—it is only a potential satellite because quite possibly, none of its neighbors flipped a head (it has no center to hook up to).

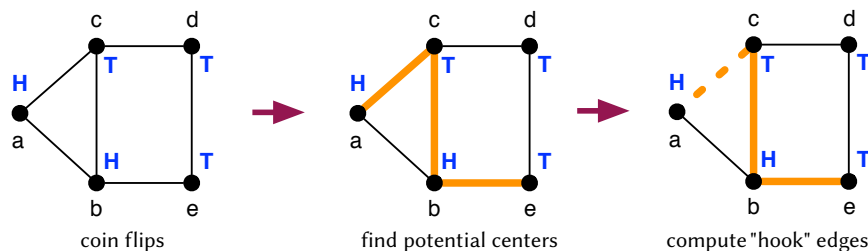
At this point, we have determined every vertex's role, but we aren't done: for each satellite vertex, we still need to decide which center it will join. For our purposes, we're only interested in ensuring that the stars are disjoint, so it doesn't matter which center a satellite joins. We will make each satellite choose any center in its set of neighbors arbitrarily. In code, we have the following:

```

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  $C = \{v \mapsto \text{coinFlip}(v, r) : v \in V\}$ 
8    % select edges that go from a tail to a head
9    val  $TH = \{(u, v) \in E \mid \neg C_u \wedge C_v\}$ 
10   % make mapping from tails to heads, removing duplicates
11   val  $P = \cup_{(u,v) \in TH} \{u \mapsto v\}$ 
12   % remove vertices that have been remapped
13   val  $V' = V \setminus \text{domain}(P)$ 
14 in ( $V', P$ ) end

```

How does this code actually work? We will go through a small example. Consider the following graph, where the coin flips turned up as indicated in the figure.



First, we flip a coin for each vertex. Once the coins are tossed, the rest of the process is completely deterministic. In the second step, we identify all edges that connect a satellite node with a center; these are the bold edges in the middle figure. In the code, these are edges found in TH . Notice that some potential satellites (nodes that flipped tails) are adjacent to multiple centers (nodes that flipped heads). For example, node c is adjacent to nodes a and b , both of which got heads. A node like this will have to choose which center to join. This is decided in the step where we compute the mapping P —this is a mapping from every tail node (satellite) present in TH to a center. In this example, c is hooked up with b , leaving a a center without any satellite.

Before we move on, let's examine P and $\text{domain}(P)$. First, P contains all the star edges, so $P = \{c \mapsto b, e \mapsto b\}$. This means that $\text{domain}(P)$ contains $\{c, e\}$ and therefore, $V' = V \setminus \text{domain}(P) =$

$\{a, b, d\}$. In general, V' is the set of vertices whose coin flipped heads or whose coin flipped tails but didn't have a neighboring center.

Analysis of Star Contraction. When we contract these stars found by `starContract`, each star becomes one vertex, so the number of vertices removed is the size of P . In expectation, how big is P ? The following lemma shows that on a graph with n non-isolated nodes, the size of P —or the number of vertices removed in one round of star contraction—is at least $n/4$.

Lemma 1.2. For a graph G with n non-isolated 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. Consider any non-isolated vertex $v \in V(G)$. Let H_v be the event that a vertex v comes up heads, T_v that it comes up tails, and R_v that $v \in \text{domain}(P)$ (i.e, it is removed). By definition, we know that a non-isolated 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, $\Pr[R_v] \geq \Pr[T_v] \Pr[H_u] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

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

since we have n vertices that are non-isolated. □

Using `ArraySequence` and `STArraySequence`, we can implement `starContract` reasonably efficiently in $O(n + m)$ work and $O(\log n)$ span for a graph with n nodes and m edges.

2 Connectivity Using Star Contraction

How would we use star contraction to solve the connectivity problem? It is instructive to start with a slightly simpler problem. We'll look at the problem of determining the number of connected components in a graph. One nice property of graph contraction is that each contraction step preserves the number of connected components, so if we keep on contracting, we will eventually be down to one vertex for every connected component in the graph. Consider the following code:

```

1 fun numComponents((V, E), r) =
2   if |E| = 0 then |V| else
3     let
4       val (V', P) = starContract((V, E), r)
5       val E' = {(\overline{P}_u, \overline{P}_v) : (u, v) \in E | \overline{P}_u \neq \overline{P}_v}
6     in numComponents((V', E'), next(r))
7   end

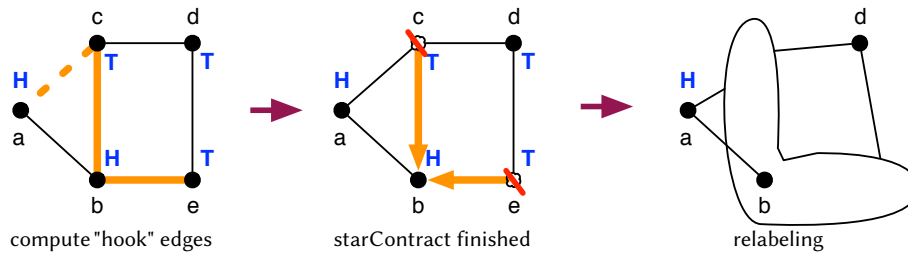
```

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).

In this code, as long as the graph is not a collection of isolated vertices, we'll run `starContract`, which will find disjoint stars in this graph. To contract these stars, we make each star center the representative node for the whole star—that is, we contract its satellites into the center. But then, we need to process the edges. For this, we construct a new edge set E' as $\{(\overline{P}_u, \overline{P}_v) : (u, v) \in E \mid \overline{P}_u \neq \overline{P}_v\}$. This dense line deserves more explanation.

For each existing edge $(u, v) \in E$, we first check if u and v belong in the same star—testing whether \overline{P}_u is the same \overline{P}_v . Only edges that have $\overline{P}_u \neq \overline{P}_v$ are kept. This is because we only retain edges that go between stars. Among the edges that we keep, we will need to relabel them because their endpoints might have disappeared. For each surviving edge (u, v) , we include in E' the edge $(\overline{P}_u, \overline{P}_v)$. That is, if a vertex is contracted into a star center, it assumes the identifier of the star center; otherwise, it keeps its identifier

This might still be confusing, so a small example is in order. We will continue with our example graph from above:



When we contract the star, vertices c and e are gone. This means that the edges (c, b) and (b, e) are internal to the star and vanish. So then, the edge (c, d) becomes (b, d) , the edge (a, c) becomes (a, b) , and the edge (d, e) becomes (d, b) .

Performance of `numComponents`. Before we move on to the connectivity problem, let's analyze the cost of this `numComponents` algorithm. Let n be the number of non-isolated vertices. Notice that once a vertex becomes isolated (due to contraction), it stays isolated until the final round (contraction only removes edges). Therefore, we have the following span recurrence (we'll look at work later):

$$S(n) = S(n') + O(\log n)$$

where $n' = n - X_n$ and X_n is the number of vertices removed (as defined earlier in the lemma about `starContract`). But $\mathbf{E}[X_n] = n/4$ so $\mathbf{E}[n'] = 3n/4$. This is a familiar recurrence, which we know solves to $O(\log^2 n)$.

As for work, ideally, we would like to show that the overall work is linear since we might hope that the size is going down by a constant fraction on each step. Unfortunately, this is not the case. Although we have shown that we can remove a constant fraction of the non-isolated 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. But 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 asymptotically. Hence, we have the following work recurrence:

$$W(n, m) \leq W(n', m) + O(n + m),$$

where n' is the remaining number of non-isolated vertices as defined in the span recurrence. This solves to $\mathbf{E}[W(n, m)] = O(n + m \log n)$. Altogether, this gives us the following theorem:

Theorem 2.1. For a graph $G = (V, E)$, `numComponents` using `starContract` graph contraction with an array sequence works in $O(|V| + |E| \log |V|)$ work and $O(\log^2 |V|)$ span.

2.1 Connectivity

We are ready to develop an algorithm for the connectivity problem. Specifically, we'll design an algorithm that on input a graph $G = (V, E)$, produce a sequence C of length $|V|$ such that $C_i = C_j$ if and only if nodes i and j are connected in G . To accomplish this, we'll augment the code we have for `numComponents`. In the base case of the `numComponents` algorithm, we have found all the components; all we have to do is when we come back out of the recursion, we have to mark the contracted node with the right label. We have the following algorithm:

```

1 fun conn((V, E), r) =
2   if |E| = 0 then {v ↦ v : v ∈ V} else
3   let
4     val (V', P) = starContract((V, E), r)
5     val E' = {(P_u, P_v) : (u, v) ∈ E | P_u ≠ P_v}
6     val C = conn((V', E'), next(r))
7   in {v ↦ C[P_v] : v ∈ V} end

```

That is, on the way out, each contracted node inherits the label of its star center. Using `STArrays`, the added step can be done in $O(n)$ work and $O(1)$ span, so the work and span of this algorithm won't increase from before. Therefore, the connectivity problem can also be solved in $O(n + m \log n)$ work and $O(\log^2 n)$ span. It's worth mentioning that with quite a bit of work, we can improve this to $O(n + m)$ work and $O(\log^2 n)$ span—and further to $O(\log n)$ span.

3 Minimum Spanning Tree

The minimum (weight) spanning tree (MST) problem is given an connected undirected graph $G = (V, E)$, where each edge e has weight $w_e \geq 0$, find a spanning tree of minimum weight (i.e., the

sum of the weights of the edges). That is to say, we are interested in finding the spanning tree T that minimizes

$$w(T) = \sum_{e \in E(T)} w_e.$$

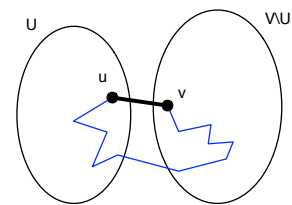
You have seen Minimum Weigh Spanning Trees in both 15-122 and 15-251. In previous classes, you went over Kruskal’s and Prim’s algorithms. At a glance, Kruskal’s and Prim’s seem to be two drastically different approaches to solving MST: whereas Kruskal’s sorts edges by weight and considers the edges in order, using a union-find data structure to detect when two vertices are in the same component and join them if not, Prim’s maintains a tree grown so far and a priority queue of edges incident on the current tree, pulling the minimum edge from it to add to the tree. The two algorithms, in fact, rely on the same underlying principle about “cuts” in a graph, which we’ll discuss next.

Light Edge Rule. The main property that underlines many MST algorithms 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 $E(U, \bar{U})$, where as is typical in literature, we write $\bar{U} = V \setminus U$. 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 following theorem states that the lightest edge across a cut is in the MST of G :

Theorem 3.1. *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.

Notice that this property was used in Kruskal’s algorithm although we won’t review its use here (15-122 covered this pretty extensively). To get a feel for how to put this property to use, we’ll review the Prim’s algorithm:

1. Pick a vertex s from G . Set $U = \{s\}$.
2. While $(|U| < n)$:
 - Pick the lightest edge uv that cross $U, V \setminus U$ (i.e. u in U, v in $V \setminus U$)
 - Add v to U and uv to the MST

It 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 weight edge $e = (u, v)$, $u \in U, v \in V \setminus U$. This is in the MST by the Theorem 3.1. 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.

As you know already, both Kruskal's and Prim's algorithms are sequential: both algorithms add in an edge or a vertex one by one and the decision at each step depends on the decisions made in previous steps.

Next time, we'll discuss a parallel MST algorithm.