Lecture 16 — Graph Contraction and Connectivity

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Umut Acar — March 19, 2013

What was covered in this lecture:

- Graph contraction and connectivity.

1 Preliminaries

Definition 1.1 (Reachability)**.** Let $G = (V, E)$ be a graph and $u, v \in V$. Vertex v is reachable from u if there is a path from u to v.

Definition 1.2 (Connectivity)**.** A graph $G = (V, E)$ is connected if for all $u, v \in V$, v is reachable from u.

Definition 1.3 (Subgraph)**.** Let $G = (V, E)$ and $H = (V', E')$ be two graphs. H is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$. H is a proper subgraph is $V' \subsetneq V$ or $E' \subsetneq E$.

Definition 1.4 (Vertex Induced Subgraph)**.** Let $G = (V, E)$ and $V' \subseteq V$. The subgraph of G induced by V is a graph $H = (V', E')$ where $E' = \{\{u, v\} \in E | u \in V', v \in V'\}$.

Definition 1.5 (Edge Induced Subgraph)**.** Let $G = (V, E)$ and $E' \subseteq E$. The subgraph of G induced by E is a graph $H = (V', E')$ where $V' = \{u | u \in \{v, w\} \in E'\}$.

Definition 1.6 ((Connected) Component)**.** Let $G = (V, E)$ be a graph. A subgraph H of G is a component of G if 1) H is connected and 2) if K is a subgraph of G and H is a proper subgraph of K then K is not connected. Thai is, H is maximally connected.

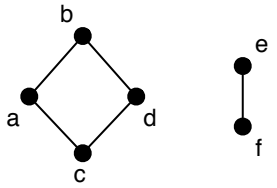
2 Graph Contraction

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 that 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. We are now going to discuss a technique called "graph contraction" that you can add to your toolbox for parallel algorithms. The technique will use randomization.

†Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

[1] In reality, there is some parallelism in both DFS and Dijkstra when graphs are dense—in particular, although vertices need to 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.

Graph Connectivity. To motivate graph contraction consider the graph connectivity problem. Two vertices are connected in an undirected graph if there is a path between them. A graph is connected if every pair of vertices is connected. The *graph connectivity* problem is to partition an undirected graph into its maximal connected subgraphs, i.e, components. By partition we mean that every vertex has to belong to one of the compenents. For example for the following graph:



graph connectivity will return the two subgraphs consisting of the vertices $\{a, b, c, d\}$ and $\{e, f\}$.

How might we solve the graph connectivity problem? We could do it with a bunch of graph searches. In particular we can start at any vertex and search all vertices reachable from it to create the first component, then move onto the next vertex and if it has not already been searched search from it to create the second component. We then repeat until all vertices have been checked. Either BFS or DFS can be used for the individual searches. This is a perfectly sensible sequential algorithm. However, it has two shortcomings from a parallelism perspective even if we use a parallel BFS. Firstly each parallel BFS takes span proportional to the diameter of the component (the longest distance between two vertices) and in general this could be as large as n. Secondly even if each component is has a small diameter, we have to iterate over the components one by one.

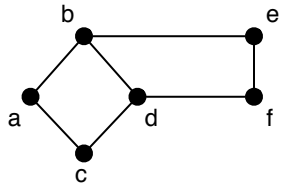
We are therefore interested in an approach that can identify the components in parallel. Furthermore the span should be independent of the diameter, and ideally polylogarithmic in $|V|$. To do this we will give up on the idea of graph search since it seems to be inherently limited by the diameter of a graph. Instead we will borrow some ideas from our algorithm for the `scan` operation. In particular we will resort to *contraction*—on each round of contraction we will shrink the size of the graph by a constant fraction and then solve the connectivity problem on the contracted graph. Contracting a graph, however, is a bit more complicated than just pairing up the odd and even positioned values as we did in the algorithm for `scan`.

This approach is called *graph contraction*. It is a reasonably simple technique and can be applied to a variety of problems, beyond just connectivity, including spanning trees and minimum spanning trees. We assume the graph is undirected. Lets start by considering a function:

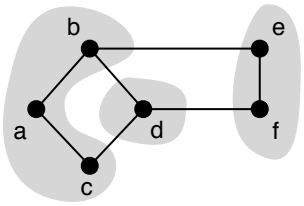
```
contract : graph → partition
```

which takes a undirected graph $G = (V, E)$ and returns a partitioning of the vertices of V such that the vertex-induced subgraph of G by each partiton is connected, and the partitions are disjoint.

For example `contract` on the following graph:

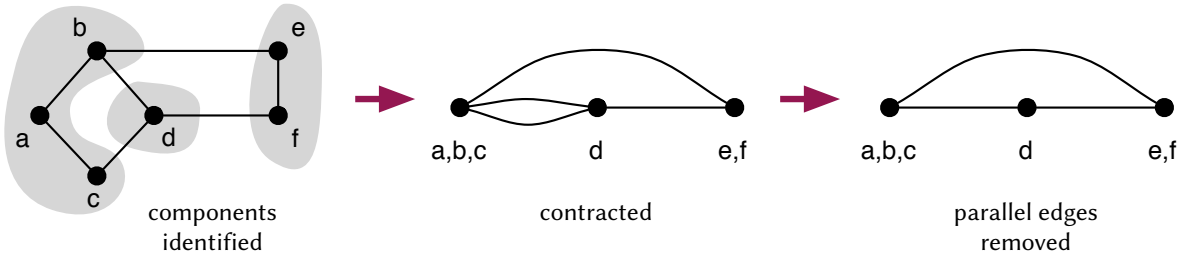


might return the partitioning $\{\{a,b,c\},\{d\},\{e,f\}\}$ as indicated by the following shaded regions:



Note that each of the three partitions is connected by edges within the partition. The `contract` function would not return $\{\{a,b,f\},\{c,d\},\{e\}\}$, for example, since the subgraph $\{a,b,f\}$ is not connected by edges within the component.

We will return to how to implement `contract`, but given that the partitions returned by `contract` are not maximal (i.e. there are still edges between them), how might we use `contract` to solve the graph connectivity problem. As we already suggested, we can apply it recursively. To do this on each round we replace each partition with a single vertex and then relabel all the edges with the new vertex name. This can be illustrated as follows:

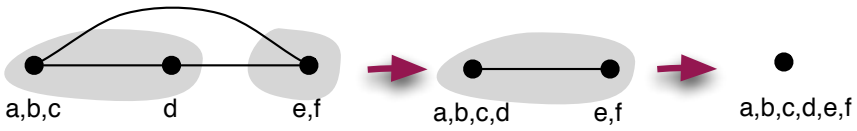


After contracting, we are left with a triangle. Note that in the intermediate step, when we join a,b,c, we create redundant edges to d (both b and c had an original edge to d). We therefore replace these with a single edge. However, depending on the particular use, in some algorithms, it is convenient to allow parallel (redundant) edges rather than going to the work of removing them. Also note that some edge are within a partition. These become self loops, which we drop.

It should be clear that if the graph actually contracts on each round (not all vertices are singleton partitions) then eventually every maximal connected component in the graph will shrink down to a single vertex. This leads to the following high-level algorithm for connectivity:

```
while there are edges left:
  contract the graph
  update the edges to use remaining vertices and remove self loops
```

Continuing with our example, after the first contraction shown earlier we might make the following two contractions in the next two rounds, at which point we have no more edges:



Note that if the input is not connected we would be left with multiple vertices when the algorithm finishes, one per component.

To make this algorithm more concrete we need some representation for the partitions. One way to do this is to use one of the original vertices of each partition as a representative and then keep a mapping from each vertex to this representative. Therefore the partition $\{\{a,b,c\},\{d\},\{e,f\}\}$ could be represented as $(\{a,d,e\},\{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\})$. With this representation first consider the following more concrete algorithm that counts the number of maximal connected components.

```
1  function numComponents((V, E), i) =
2  if |E| = 0 then |V|
3  else let
4      (V', P) = contract((V, E), i)
5      E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
6  in
7    numComponents((V', E'), i + 1)
8  end
```

The i is just the round number—as we will see it will be useful in `contract`. As in the high-level description, `numComponents` contracts the graph on each round. Each contraction on Line 4 returns the contracted vertices V' and a table P mapping every $v \in V$ to a $v' \in V'$. Line 5 then does two things. Firstly it updates all edges so that the two endpoints are in V' by looking them up in P: this is what $(P[u], P[v])$ is. Secondly it removes all self edges: this is what the filter $P[u] \neq P[v]$ does. Once the edges are updated, the algorithm recurses on the smaller graph. The termination condition is when there are no edges. At this point each component has shrunk down to a singleton vertex and we just count them with $|V|$.

To make the code return the partitioning of the connected components is not much harder, as can be seen by the following code:

```
1  function components((V, E), i) =
2  if |E| = 0 then {v ↦ v : v ∈ V}
3  else let
4      (V', P) = contract((V, E), i)
5      E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
6      P' = components((V', E'), i + 1)
7  in
8    {v ↦ P'[P[v]] : v ∈ V}
9  end
```

This code returns the partitioning in the same format as the partitioning returned by `contract`. The only difference of this code from `numComponents` is that on the way back up the recursion we update the representatives for V based on the representatives from V' returned by the recursive call. Consider our example graph. As before lets say the first `contract` returns

$$\begin{aligned} V' &= \{a, d, e\} \\ P &= \{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\} \end{aligned}$$

Since the graph is connected the recursive call to `components` will map all vertices in V' to the same vertex. Lets say this vertex is "a" giving:

$$P' = \{a \mapsto a, d \mapsto a, e \mapsto a\}$$

Now what Line 7 in the code does is for each vertex $v \in V$, it looks for v in P to find its representative $v' \in V'$, and then looks for v' in P' to find its representative in the connected component. This is implemented as $P'[P[v]]$. For example vertex f finds e from P and then looks this up in P' to find a. The final result returned by `components` is:

$$\{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto a, e \mapsto a, f \mapsto a\}$$

The base case of the `components` algorithm is to label every vertex with itself.

This leaves us with the question of how to implement `contract`. There are a three kinds of contraction we will consider:

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.

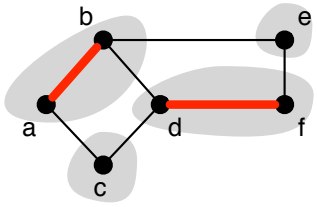
Star Contraction: One vertex of each component is identified as the center of the star and all other vertices are directly connected to it.

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

Keep in mind that the goal here is to do the contraction in parallel. Furthermore we want to contract the graph size (number of vertices) by a constant factor on each round. This way we will ensure the algorithm will run with just $O(\log n)$ rounds (contractions).

2.1 Edge Contraction

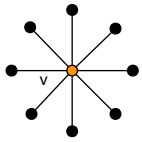
Lets consider contracting edges. How would we find a set of disjoint edges to contract? By disjoint we mean that the no two contraction edges can share a vertex. Here is an example of a set of disjoint edges on which to contract:



The edges to contract are $\{\{a,b\},\{d,f\}\}$. Note that the remaining two vertices, e and c, are left on their own.

Finding such a set of edges to contract is the problem of finding a *vertex matching*, i.e. we are trying to match every vertex with another vertex (monogamously). It turns out this can be done in parallel relatively easily by having every edge pick a random priority in the range $[0,1]$ and then choosing an edge if it has the highest priority on both the vertices it is incident on. Before we delve in further, however, we note that there is a problem with edge contraction. Consider the following kind of graph:

Definition 2.1 (Star)**.** A *star* graph $G = (V,E)$ is an undirected graph with a center vertex $v \in V$, and a set of edges $E = \{\{v,u\} : u \in V \setminus \{v\}\}$.



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 graph, will edge contraction work well on this? How many edges can contract on each round. It is not difficult to convince ourselves that on a star graph with $n+1$ vertices—1 center and n "satellites"—any edge contraction algorithm will take $\Omega(n)$ rounds. We therefore move on.

2.2 Star Contraction

We now consider a more aggressive form of contraction that can be used to contract subgraphs which are stars in a single round. The idea is to combine the star center with all its "satellites" all at once. 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?* Again, disjoint means that each vertex belongs to at most one partition, i.e. star. As an example, in the graph below (left), we can find 2 disjoint 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 will 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.

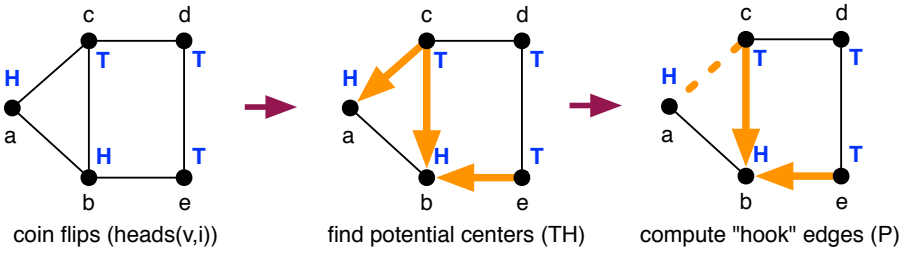
As usual, we'll start by flipping a coin for each vertex. If it comes up heads, that vertex is a star center. And if it comes up tails, then it will be a potential satellite—it is only a potential satellite because quite possibly, none of its neighbors flipped a head (it has no center to hook up to).

At this point, we have determined every vertex's potential 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.

Before describing the code the code, we need to say a couple words about the source of randomness. What we will assume is for each vertex we have a potentially infinite sequence of random coin flips and that we can access the i^{th} one with the function $\texttt{heads}(v,i) : \texttt{vertex} \times \texttt{int} \to \texttt{bool}$ that returns true if the i^{th} flip on vertex v is heads and false otherwise. You can think of it as having flipped all the coins ahead of time, stored a sequence of flips for each vertex, and now you are using `heads`to access the i^{th} one. Since most machines don't have true sources of randomness, in practice this can be implemented with a pseudorandom number generator or even with a good hash function. We are now ready for the code for star contraction:

```
1  % requires: an undirected graph G = (V, E) and round number i
2  % returns: V' = remaining vertices after contraction,
3  %           P = mapping from V to V'
4  function starContract(G = (V, E), i) =
5  let
6    % select edges that go from a tail to a head
7      TH = {(u, v) ∈ E | ¬heads(u, i) ∧ heads(v, i)}
8    % make mapping from tails to heads, removing duplicates
9      P = ∪_{(u,v)∈TH} {u ↦ v}
10   % remove vertices that have been remapped
11     V' = V \ domain(P)
12   % Map remaining vertices to themselves
13     P' = {u ↦ u : u ∈ V'} ∪ P
14 in (V', P') end
```

mall example. Consider the following graph, where the coin flips turned up as indicated in the figure.



The function $\texttt{heads}(v,i)$ on round i gives a coin flip for each vertex, which are shown on the left. Line 7 selects the edges that go from a tail to a head, which are shown in the middle as arrows and correspond to the set $TH = \{(c,a),(c,b),(e,b)\}$. Notice that some potential satellites (vertices that flipped tails) are adjacent to multiple centers (vertices that flipped heads). For example, vertex c is adjacent to vertices a and b, both of which got heads. A vertex like this will have to choose which

center to join. This is sometimes called "hooking" and is decided on Line 9, which removes duplicates for a tail using union, giving $P = \{c \mapsto b, e \mapsto b\}$. In this example, c is hooked up with b, leaving a a center without any satellite.

Line 11 takes the vertices V and removes from them all the vertices in the domain of P, i.e. those that have been remapped. In our example $\texttt{domain}(P) = \{c, e\}$ so we are left with $V' = \{a, b, d\}$. In general, V' is the set of vertices whose coin flipped heads or whose coin flipped tails but didn't have a neighboring center. Finally we map all vertices in V' to themselves and union this in with the hooks giving $P' = \{a \mapsto a, b \mapsto b, c \mapsto b, d \mapsto d, e \mapsto b\}$.

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 vertices, the size of P—or the number of vertices removed in one round of star contraction—is at least $n/4$.

Lemma 2.2. *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,* $\mathbf{E}\left[X_n\right] \geq n/4$.

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 $v \in \texttt{domain}(P)$ (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, $\mathbf{Pr}\left[R_v\right] \geq \mathbf{Pr}\left[T_v\right]\mathbf{Pr}\left[H_u\right] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

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

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

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

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 vertices and m edges.

2.3 Returning to Connectivity

Now lets analyze the cost of the algorithm for counting the number of connected components we described earlier when using star contraction for `contract`. 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 $\mathbf{E}\left[X_n\right] = n/4$ so $\mathbf{E}\left[n'\right] = 3n/4$. 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 round. 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 round, 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 rounds:

round	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 2.3. *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.*

2.4 Tree Contraction

Tree contraction takes a set of disjoint trees and contracts them. One way to do this is by repeated star contraction. The advantage of tree contraction over general graph contraction is that the number of edges on each step is at most one less than the number of remaining vertices. Therefore the number of edges must go down geometrically from step to step, and the overall cost of tree contraction is $O(m)$ work and $O(\log^2 n)$ span using an array sequence.

3 Spanning Trees and Forests

Recall that an undirected graph is a forest if it has no cycles and is a tree if it has no cycles and is connected. A tree on n vertices always has exactly $n - 1$ edges, and a forest has at most $n - 1$ edges. A *spanning tree* of an undirected connected graph $G = (V, E)$ is a tree $T = (V, E')$ where $E' \subseteq E$. A *spanning forest* of a graph $G = (V, E)$ is the union of spanning trees on its connected components. We are interested in the spanning forest problem, which is to find a spanning forest for a given undirected graph (the spanning tree is just the special case when the input graph is connected).

It turns out that a spanning forest of a graph G can be generated from our connectivity algorithm. In particular all we need to do is keep track of all the edges that we use to hook, and return the union of these edges. We will see this in more detail as we cover minimum spanning trees, our next topic.