Lecture 16 — Graph Contraction

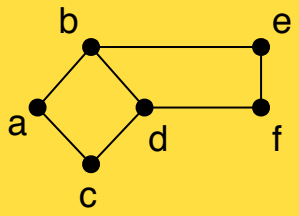
Parallel and Sequential Data Structures and Algorithms, 15-210 (Qatar-Spring 2014)

Lectured by Kemal Oflazer — March 18, 2014

1 Preliminaries

We will start by reviewing some graph terminology that will be helpful for the material in this lecture. We will use the following graph as an example.

Example 1.1. *We will use the following undirected graph as an example.*



Question 1.2. *Do you recall how we characterize reachability of a vertex v from another vertex u? For example, is f reachable from a?*

Definition 1.3. 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.

Based on reachability, we can define connectivity in graphs.

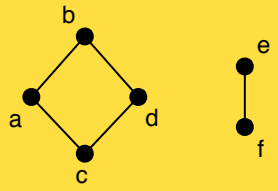
Definition 1.4. A graph $G = (V, E)$ is connected if for all $u, v \in V$, v is reachable from u.

For example, our example is connected.

Question 1.5. *Can you think of a way to disconnect the graph?*

†Lecture notes by Umut Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, with additional edits by Kemal Oflazer

Example 1.6. *You can disconnect the graph by deleting two enges for example* (d, f) *and* (b, e).



In this class, we will need a way to refer to part of a graph, which we call a subgraph.

Question 1.7. *Any intuition about how we may define a subgraph?*

We can take any subsets of edges and vertices as long as the result is a well defined graph.

Definition 1.8 (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 $E' \subsetneq E$.

There are thus many subgraphs of a graph.

Question 1.9. *How many subgraph would a graph with n vertices and m edges have?*

This is not an easy question. We cannot just take arbitrary subsets of vertices and edges because the resulting subsets must define a graph. For example, we cannot have an edge between two non-existing vertices.

When talking about subgraphs, it is often helpful to consider subgraphs that are maximal in same way. For example, we refer to a subgraph of a graph as a component if it is connected and it cannot be made larger without breaking connectivity.

Definition 1.10 ((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. That is, H is a maximally connected subgraph.

Question 1.11. *How many connected components does our graphs have?*

Our first example graph has one connected component (hence it is connected). The second has two connected components.

Determining connectivity and the connected components of a graph are interesting computations. When performing such computations, we will often talk about subgraphs of a graph that are naturally

induced or implied by a set of vertices or edges.

Definition 1.12 (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 $E' = \{\{u, v\} \in E | u \in V', v \in V'\}$.

Question 1.13. *In our running example, what would be the subgraph induced by the vertices* $\{a, b\}$*? How about* $\{a, b, c, d\}$*?*

Symmetrically, we can consider a subgraph solely defined by a set of edges.

Question 1.14. *Can you imagine how we might define such an "edge induced" subgraph?*

We have to take all the vertices to make the resulting object a proper graph. In other words, we take the vertices that we absolutuly have to.

Definition 1.15 (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 $V' = \{u | u \in \{v, w\} \in E'\}$.

2 Graph Contraction

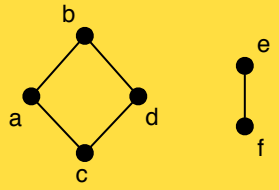
Until recently, we have been talking mostly about techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of them are easy to parallelize while others are not. For example, we saw that there is parallelism in BFS because each level can be explored in 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.

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.

Definition 2.1. The *graph connectivity* problem is to partition an undirected graph into its components (maximal connected subgraphs)

By partition we mean that every vertex has to belong to one of the compenents.

Example 2.2. *For the following graph:*



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

Question 2.3. *Can you think of a way of solving the graph-connectivity problem?*

One way would be to use graph search. For example, 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.

[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.

Question 2.4. *What kinds of algorithms can you use to perform the searches?*

Either BFS or DFS can be used for the individual searches.

Question 2.5. *Would these approaches yield good parallelism? What would be the span of the algorithm?*

These algorithms are perfectly sensible sequential algorithms. But they are not good parallel algorithms. Recall that DFS has linear span.

Question 2.6. *How about BFS? Do you recall the span of BFS?*

BFS takes span proportional to the diameter of the graph. In the context of our algorithm the span would be the diameter of the component (the longest distance between two vertices).

Question 2.7. *How large the diameter of a component can be? Can you give an example?*

The diameter of a component can be as large as n. An an example consider a "chain" of vertices.

Question 2.8. *How about in cases when the diameter is small, for example when the graph is just a disconnected collection of edges.*

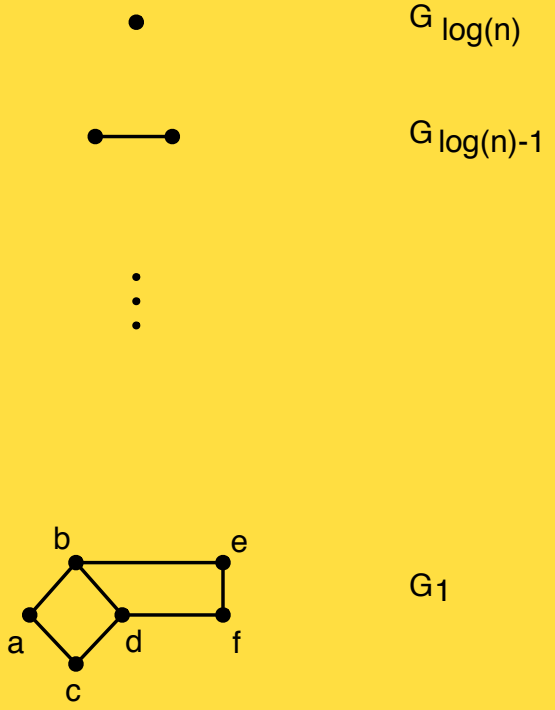
In that case, we have to iterate over the components one by one. Thus the span in the worst case can be linear in the number of components.

Contraction hierarchies. We are interested in an approach that can identify the components in parallel. We also wish to come with an algorithm whose span is 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 use *contraction*.

The idea contraction is to shrink the graph by a constant fraction, while respecting the connectivity of the graph. We can then solve the problem on the smaller, contracted graph and from that result compute the result for the actual graph.

Example 2.9 (A contraction hierarchy)**.** *The crucial poin to remember is that we want to build the contraction hierarchy in a way that respects and reveals the connectivity of the vertices in a hierarchical way. Vertices that are not connected should not become connected and vice versa.*



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.

As an anology, you can think of graph contraction as a way of viewing a graph at different levels of detail. For example, if you want to drive from Pittsburgh to San Francisco, you don't concern yourselves with all the little towns on the road and all the roads in between them. Rather you think of highways and the interesting places that you may want to visit. As you think of the graph at different levels, however, you certainly don't want to ignore connectivity. For example, you don't want to find yourself hopping on a ferry to the UK on your way to San Francisco.

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`.

Question 2.10. *Any ideas about how we might contract a graph?*

When contracting a graph, we want to take subgraphs of the graph and represent them with what we might call *super-vertices*, adjusting the edges accordingly. By selecting the subgraphs carefully, we will maintain the connectivity and at the same time shrink the graph geometrically (by a constant

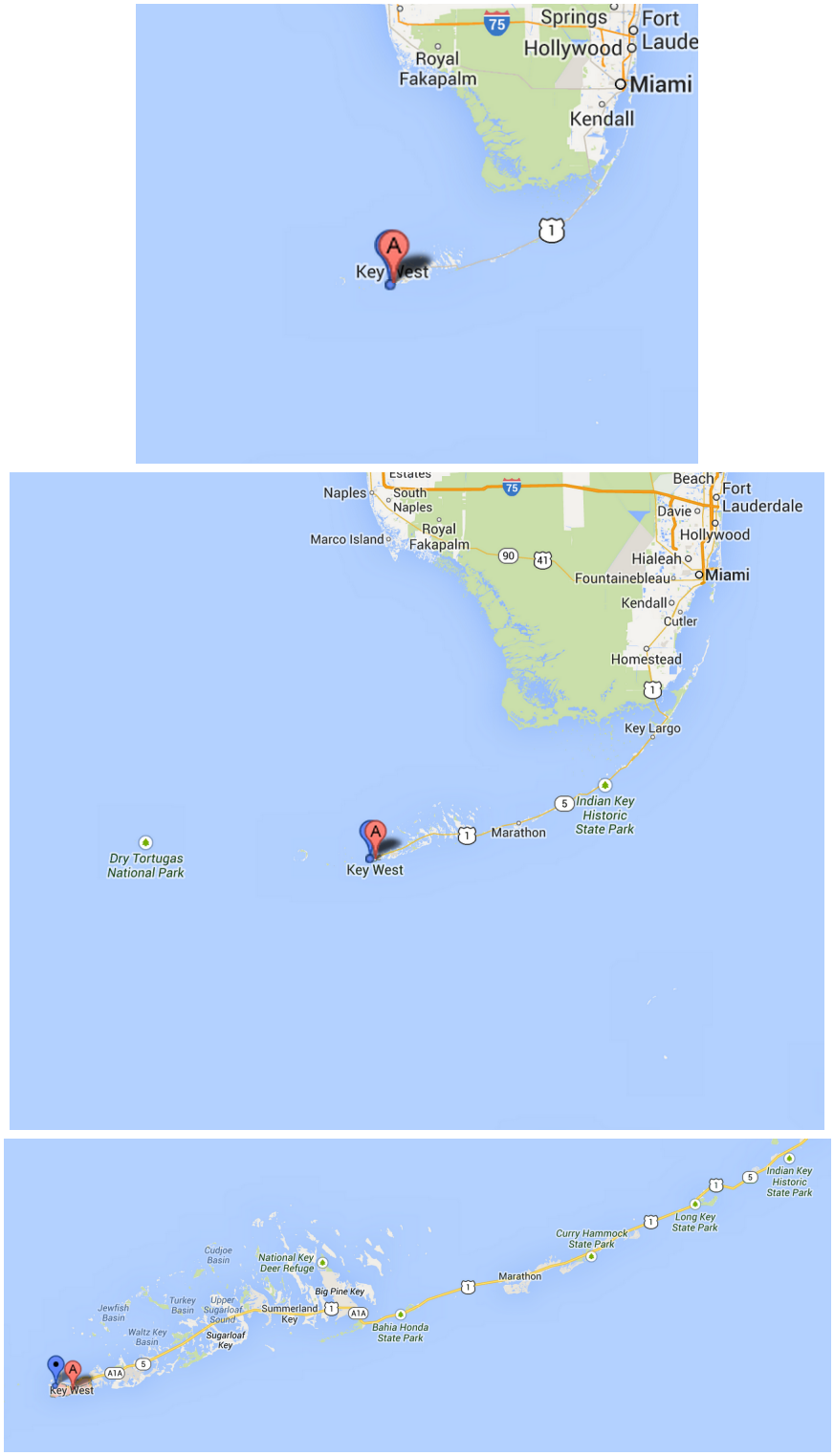


Figure 1: An illustration of graph contraction using maps. The road to key west at three different zoom levels. At first (top) there is just a road (an edge). We then see that there is an island (Marathon) and in fact two highways (1 and 5). Finally, we see more of the islands and the roads in between.

factor). We then treat the super-vertices as ordinary vertices and repeat the process until there are no more edges.

The key question is what kind of subgraphs we should take. Let's first ignore the efficiency issue.

Question 2.11. *Can we contract subgraph that are disconnected?*

By replacing each subgraph with a vertex, we are essentially saying that the vertices in the subgraph are all one and can be reachable from each other. Therefore, the subgraphs that we choose should connected, because otherwise we will not be consistent with connectivity of the graph.

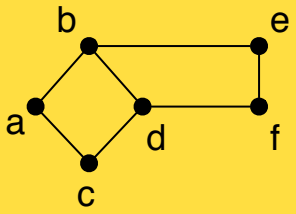
Question 2.12. *Can the subgraphs that we choose to contract overlap with each other, that is share a vertex or an edge.*

Although this might work, for our purposes, we will choose disjoint subgraphs. Suppose now that we are given the function:

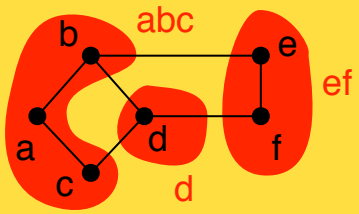
$$\texttt{partitionGraph} : \texttt{graph} \to \texttt{vertices} * \texttt{partition}$$

which takes a undirected graph $G = (V, E)$ and set of super vertices and a partition specified as a mapping between the vertices of V and supervertices such that the vertex-induced subgraph of G by each partition is connected, and the partitions are disjoint.

Example 2.13. *The function* `partitionGraph` *on the following graph:*



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



We name the super vertices as abc, d, *and* ef.
Note that each of the three partitions is connected by edges within the partition. The `partitionGraph` *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 can now write the pseudocode for graph contraction as follows.

Pseudo Code 2.14. *The pseudocode for graph contraction.*

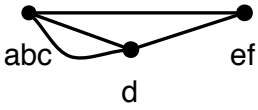
```
1  fun contractGraph((V,E), i) =
2  if |E| = 0 then (V,E)
3  else let
4      (V',P) = partitionGraph((V,E), i)
5      E' = {(P[u],P[v]) : (u,v) ∈ E | P[u] ≠ P[v]}
6  in
7    contractGraph((V',E'), i+1)
8  end
```

The i is just the round number—as we will see it will be useful in `partitionGraph`. As in the high-level description, `contractGraph` contracts the graph in each round. Each contraction on Line 4 returns the set of super vertices V' and a table P mapping every $v \in V$ to a $v' \in V'$.

Line 5 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.

An example graph contraction. We proceed in rounds. In each round we pick a partitioning and replace each partition with a single vertex and then relabel all the edges with the new vertex name. 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). This is shown below.



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.
After first contraction shown earlier we might make the illustrated additional 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.

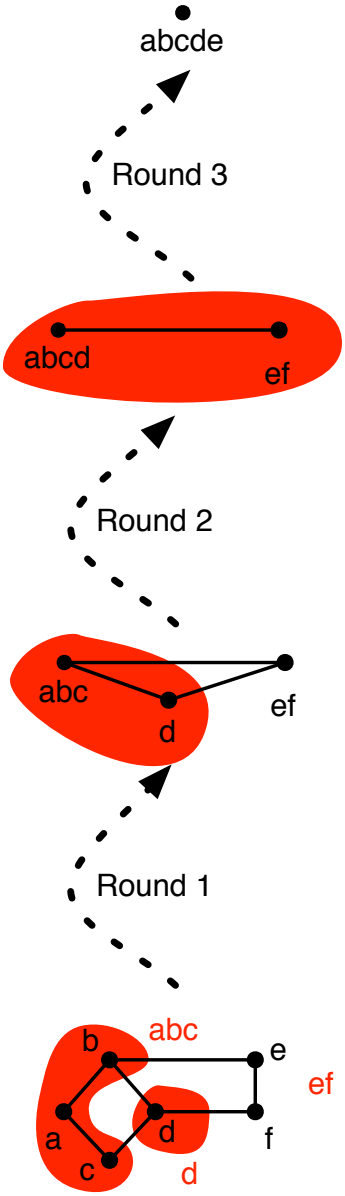


Figure 2: An example graph contraction. The first step also illustrates the elimination of parallel edges.

Naming super vertices. In our example, we gave fresh names to super vertices. It is often more convenient to pick a representative from each partiton as a super vertex. We can then represent `partition` as a mapping from each vertex to its representative (super vertex). For example, we can write the partition $\{\{a,b,c\},\{d\},\{e,f\}\}$ as $(\{a,d,e\},\{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\})$.

Counting the number of components. Now that we have developed graph contraction, we can use it to count the number of components in a graph.

Pseudo Code 2.15 (Number of components).

```
1  fun numComponents((V, E), i) =
2  if |E| = 0 then |V|
3  else let
4      (V', P) = partitonGraph((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 only difference between this code and `contractGraph` is that it returns the number of vertices when contraction completes.

Computing the components. It turns out we can modify the code slightly to compute the components themselves instead of returning their count.

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

Pseudo Code 2.16 (Components).

```
1  fun components((V, E), i) =
2  if |E| = 0 then {v ↦ v : v ∈ V}
3  else let
4      (V', P) = partitonGraph((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 `partitionGraph`. 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 `partitionGraph` 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.

Different forms of graph contraction. In our discussion so far, we have not specified the behavior of `partitionGraph` carefully. Depending on how we partition the graph, we are going to consider three different kinds of graph contraction.

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

Star Contraction: We contract subgraphs induced by vertices of a *star systems* or *stars* for short by using a partition function that returns stars. Each start consists of a vertex (a center or a star) and satellites attached to it with a single edge.

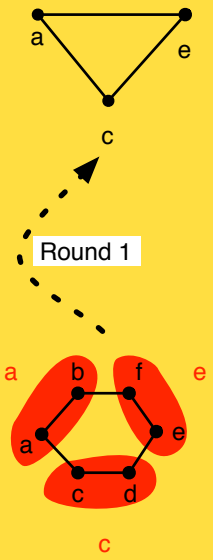
Tree Contraction: Generalizing star contraction, disjoint trees within the graph are identified and tree contraction is performed on these trees.

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

In edge contraction, the idea is to partition the graph into components consisting of at most two vertices and the edge between them (when possible). We then contract the edges in order to shrink the graph into a smaller graph.

Example 2.17. *The figure below shows an example edge contraction.*



In the above example, we were able to reduce the size of the graph by a factor of two by contracting along the selected edges. The key question is how to select such edges.

Question 2.18. *Can you describe an algorithm for selecting the disjoint set of edges that the* `partitonGraph` *function would return?*

One way would be to start at a vertex and pair in up with one of its neighbors. We then continue as much as we can. Eventually we will be left with a number of vertices that we cannot pair up with anything else and some paired up vertices. We return that as a partition.

The problem with that algorithm is that it is completely sequential.

Question 2.19. *Can you think of a way of partitioning the graph in parallel?*

For a parallel algorithm, we need to be able to make local decisions at each vertex. So what we can do is for each vertex, we can pick one of its neighbors to pair up with.

Question 2.20. *What is the problem with this approach?*

The problem is that it is not possible to ensure disjointness. We need a way to *break the symmetry*

that arises when two vertices try to pair up with the same vertex.

Question 2.21. *Can you think of a way to use randomization?*

We can use randomization to make sure that if two vertices are paired, then their neighbors do not pair with them. The idea is to flip a coin for each edge and pick that edge if the edge (u, v) flips heads and all the edges incident on u and v flip tails.

Question 2.22. *Can you see how many vertices we would pair up in our cycle graph example?*

Let R_e be an indicator random variable denoting whether e is selected or not, that is $R_e = 1$ if e is selected and 0 otherwise. Now the expectation $E[R_e] = \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$.

Thus summing over all edges, we conclude that expected number of edges deleted is $\frac{m}{8}$ (note, $m = n$). This is great because it allows us to shrink the cycle graph to a single vertex in logarithmic number of rounds. Indeed, this technique leads to an algorithm with linear work and $O(\log^2 n)$ span.

Question 2.23. *Can you think of a way to improve the expected number of edges contracted?*

We can further improve the bound by letting each edge pick a number for example between 0 and 1 and select and edge if it is the local maximum, i.e., it picked the highest number among all the edges incident on its end points. This increases the expected number of edges contracted to $\frac{m}{3}$.

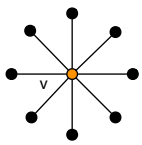
Remark 2.24. Finding a set of independent edges to contract is the problem of finding a *vertex matching*, i.e. we are trying to match every vertex with another vertex (monogamously).

Question 2.25. *So far in our example, we have considered a simple cycle graph. Do you think this technique would work as effectively for arbitrary graphs?*

Edge contraction does not work with general graphs. The problem is that if the vertex that an edge is incident on has high degree, then it is highly unlikely for the vertex to be picked. In fact, among all the edges incident an a vertex only one can be picked for cantraction. Thus in general, using edge contraction, we can shrink the graph only by a small amount.

As an example, consider a star graph:

Definition 2.26 (Star)**.** A *star* graph $G = (V, E)$ is an undirected graph with a center vertex $v \in V$, and a set of edges $E = \{\{v, u\} : u \in V \setminus \{v\}\}$.



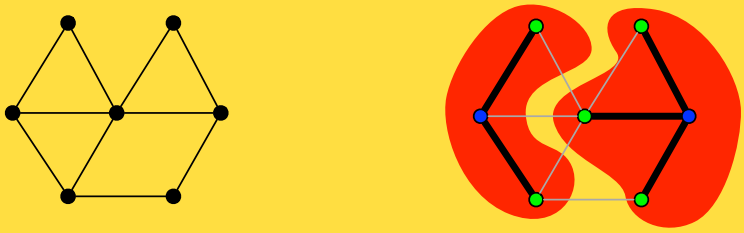
In words, a star graph is a graph made up of a single vertex v in the middle (called the center) and all the other vertices hanging off of it; these vertices are connected only to the center. Notice that a star graph is in fact a tree. If rooted at the center, it has depth 1, which is a super shallow 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. Star contraction, considered next solves this problem.

3 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 partition the graph into a set of star graphs, consisting of a center and satellites, and contract each star into a vertex in one round.

Example 3.1. *In the graph below (left), we can find 2 disjoint stars (right). The centers are colored blue and the neighbors are green.*



The key question again is the determination of the partition.

Question 3.2. *How can we find disjoint stars?*

We can construct stars sequentially as we did in edge contraction but we want to have a parallel algorithm that makes local decisions. As in edge contraction when making local decisions, we need a way to break the symmetry between two vertices that want to become the center of the star.

Question 3.3. *Can you think of a randomized algorithm for selecting stars (centers and satellites)?*

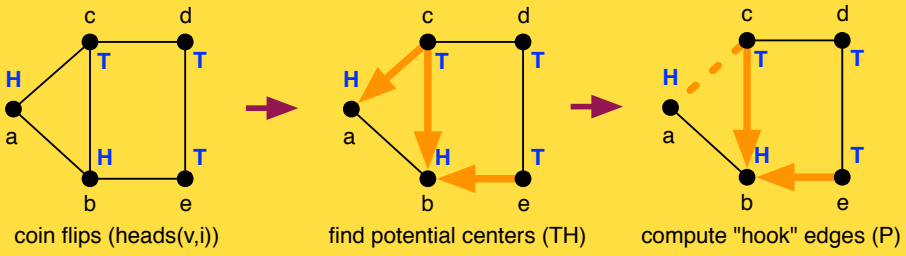
We can use coin flips to decide which vertices will be centers and which ones will be satellites. We then decide how to pair up each satellite with a center. To determine the centers, we will flip a coin for each vertex. If it comes up heads, that vertex is a star center. And if it comes up tails, then 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.

Question 3.4. *Can you come up with an algorithm choosing the satellites for each center? Do we need randomization?*

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.

Example 3.5. *An example star partition. Coin flips turned up as indicated in the figure.*



Before describing the code for partitioning a graph into stars, 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 $\texttt{heads}(v,i) : \texttt{vertex} \times \texttt{int} \to \texttt{bool}$.
The function 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:

Pseudo Code 3.6 (Star Partition)**.**

```
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  fun starPartition(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
```

In our example graph, 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 3.7. *For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by* $\texttt{starContract}(G, r)$*. 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 $v \in \texttt{domain}(P)$ (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, $\mathbf{Pr}[R_v] \geq \mathbf{Pr}[T_v]\mathbf{Pr}[H_u] = 1/4$. 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. □

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

Cost Specification 3.8 (Star Contraction). *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.*

3.1 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 $\mathbf{E}[X_n] = n/4$ 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 3.9. *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.*

3.2 Tree Contraction

Tree contraction takes a set of disjoint trees and contracts them.

When contracting a tree, we can use star contraction as before but we can show a tighter bound on work because the number of edges decrease geometrically (the number of edges in a tree is bounded by the number of 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.

When contracting a graph, we can also use a partition function that select disjoint trees (star contraction is a special case). As in star contraction, we would pick the subgraph induced by the vertices in each tree and contract them.

4 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.