Chapter 14

Graph Contraction and Connectivity

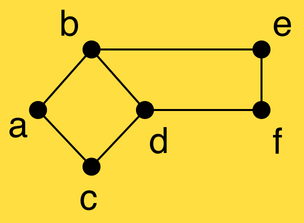
So far we have mostly covered techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of them are easy to parallelize while others are not. For example, we saw that BFS has some parallelism since each level can be explored in parallel, but there was no parallelism in DFS [1] There was also limited parallelism in Dijkstra's algorithm, but there was plenty of parallelism in the Bellman-Ford algorithm. In this chapter we will cover a technique called "graph contraction" that was specifically designed to be used in parallel algorithms and allows us to get polylogarithmic span for certain graph problems.

Also, so far, we have been only described algorithms that do not modify a graph, but rather just traverse the graph, or update values associated with the vertices. As part of graph contraction, in this chapter we will also study techniques for restructure graphs.

14.1 Preliminaries

We start by reviewing and defining some new graph terminology. We will use the following graph as an example.

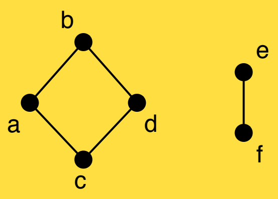
Example 14.1. *We will use the following undirected graph as an example.*



[1] In reality, there is parallelism in DFS when graphs are dense—in particular, although vertices need to visited sequentially, with some care, the edges can be processed in parallel.

Recall that in a graph (either directed or undirected) a vertex v is reachable from a vertex u if there is a path from u to v. Also recall that an undirected graph is connected if all vertices are reachable from all other vertices. Our example is connected.

Example 14.2. *You can disconnect the graph by deleting two edges, for example* (d, f) *and* (b, e).



When working with graphs it is often useful to refer to part of a graph, which we will call a subgraph.

Question 14.3. *Any intuition about how we may define a subgraph?*

A subgraph can be defined as any subsets of edges and vertices as long as the result is a well defined graph, and in particular:

Definition 14.4 (Subgraph)**.** *Let* $G = (V, E)$ *and* $H = (V', E')$ *be two graphs.* H *is a subgraph of if* $V' \subseteq V$ *and* $E' \subseteq E$.

There are many subgraphs of a graph.

Question 14.5. *How many subgraph would a graph with* n *vertices and* m *edges have?*

It is hard to count the number of possible subgraphs since 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.

One of the most standard subgraphs of an undirected graph are the so called connected components of a graph.

Definition 14.6 ((Connected) Component)**.** *Let* $G = (V, E)$ *be a graph. A subgraph* H *of* G *is a connected component of* G *if it is a maximal connected subgraph of* G.

In the definition "maximal" means we cannot add any more vertices or edges from G without disconnecting it. In general when we say an object is a maximal "X", we mean we cannot add any more to the object without violating the property "X".

Question 14.7. *How many connected components do our two graphs have?*

Our first example graph has one connected component (hence it is connected), and the second has two. It is often useful to find the connected components of a graph, which leads to the following problem:

Definition 14.8 (The Graph Connectivity (GC) Problem)**.** *Given an undirected graph $G = (V, E)$ return all of its connected components (maximal connected subgraphs).*

When talking about subgraphs it is often not necessary to mention all the vertices and edges in the subgraph. For example for the graph connectivity problem it is sufficient to specify the vertices in each component, and the edges are then implied—they are simply all edges incident on vertices in each component. This leads to the important notion of induced subgraphs.

Definition 14.9 (Vertex-Induced Subgraph)**.** *The subgraph of $G = (V, E)$ induced by $V' \subseteq V$ is the graph $H = (V', E')$ where $E' = \{\{u, v\} \in E \mid u \in V', v \in V'\}$.*

Question 14.10. *In Example 14.1 what would be the subgraph induced by the vertices $\{a, b\}$? How about $\{a, b, c, d\}$?*

Using induced subgraphs allows us to specify the connected components of a graph by simply specifying the vertices in each component. The connected components can therefore be defined as a partitioning of the vertices. A partitioning of a set means a set of subsets where all elements are in exactly one of the subsets.

Example 14.11. *Connected components on the graph in Example 14.2 returns the partitioning $\{\{a, b, c, d\}, \{e, f\}\}$.*

When studying graph connectivity sometimes we only care if the graph is connected or not, or perhaps how many components it has.

In graph connectivity there are no edges between the partitions, by definition. More generally it can be useful to talk about partitions of a graph (a partitioning of its vertices) in which there can be edges between partitions. In this case some edges are *internal edges* within the induced subgraphs and some are *cross edges* between them.

Example 14.12. *In Example 14.1 the partitioning of the vertices $\{\{a, b, c\}, \{c\}, \{e, f\}\}$ defines three induced subgraphs. The edges $\{a, b\}$, $\{a, c\}$, and $\{e, f\}$ are internal edges, and the edges $\{c, d\}$, $\{b, d\}$, $\{b, e\}$ and $\{d, f\}$ are cross edges.*

14.2 Graph Contraction

We now return to the topic of the chapter, which is graph contraction. Although graph contraction can be used to solve a variety of problems we will first look at how it can be used to solve the graph connectivity problem described in the last section.

Question 14.13. *Can you think of a way of solving the graph-connectivity problem?*

One way to solve the graph connectivity problem is 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.

Question 14.14. *What kinds of algorithms can you use to perform the searches?*

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

Question 14.15. *Would these approaches yield good parallelism? What would be the span of the algorithm?*

Using BFS and DFS lead to perfectly sensible sequential algorithms for graph connectivity, but they are not good parallel algorithms. Recall that DFS has linear span.

Question 14.16. *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 a component (the longest distance between two vertices).

Question 14.17. *How large can the diameter of a component be? Can you give an example?*

The diameter of a component can be as large as $n - 1$. A "chain" of n vertices will have diameter $n - 1$

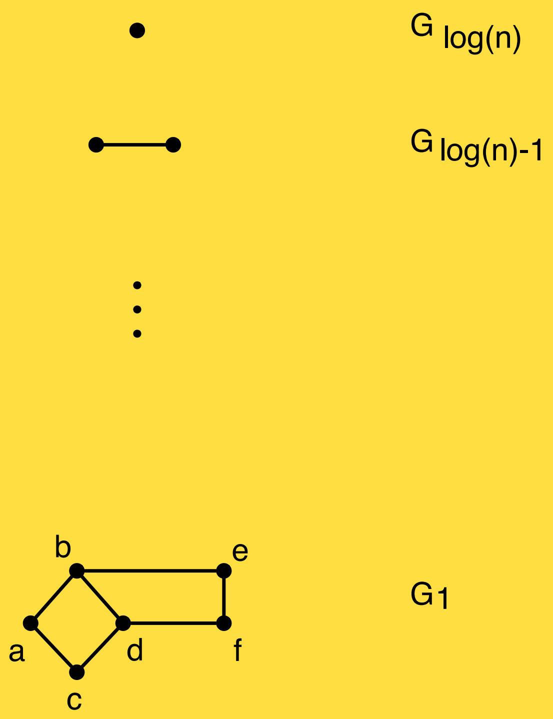
Question 14.18. *How about in cases when the diameter is small, for example when the graph is just a disconnected collection of edges.*

Even if the diameter of each component is small, we might 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 develop an algorithm whose span is independent of the diameter, and ideally polylogarithmic in $|V|$. To do this let's 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 graph 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 14.19 (A contraction hierarchy)**.** *The crucial point 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 analogy, 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 do not 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 14.20. *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 call *supervertices*, 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 factor). We then treat the supervertices as ordinary vertices and repeat the process until there are no more edges.

The key question is what kind of subgraphs should we contract and represent as a single vertex. Let's first ignore efficiency for now and just consider correctness.

Question 14.21. *Can we contract subgraph that are disconnected?*

We surely should not take disconnected subgraphs since by replacing each subgraph with a vertex, we are essentially saying that the vertices in the subgraph are connected. Therefore, the subgraphs that we choose should be connected.

Question 14.22. *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. We therefore will just consider partitions or our input graph where each partition is connected.

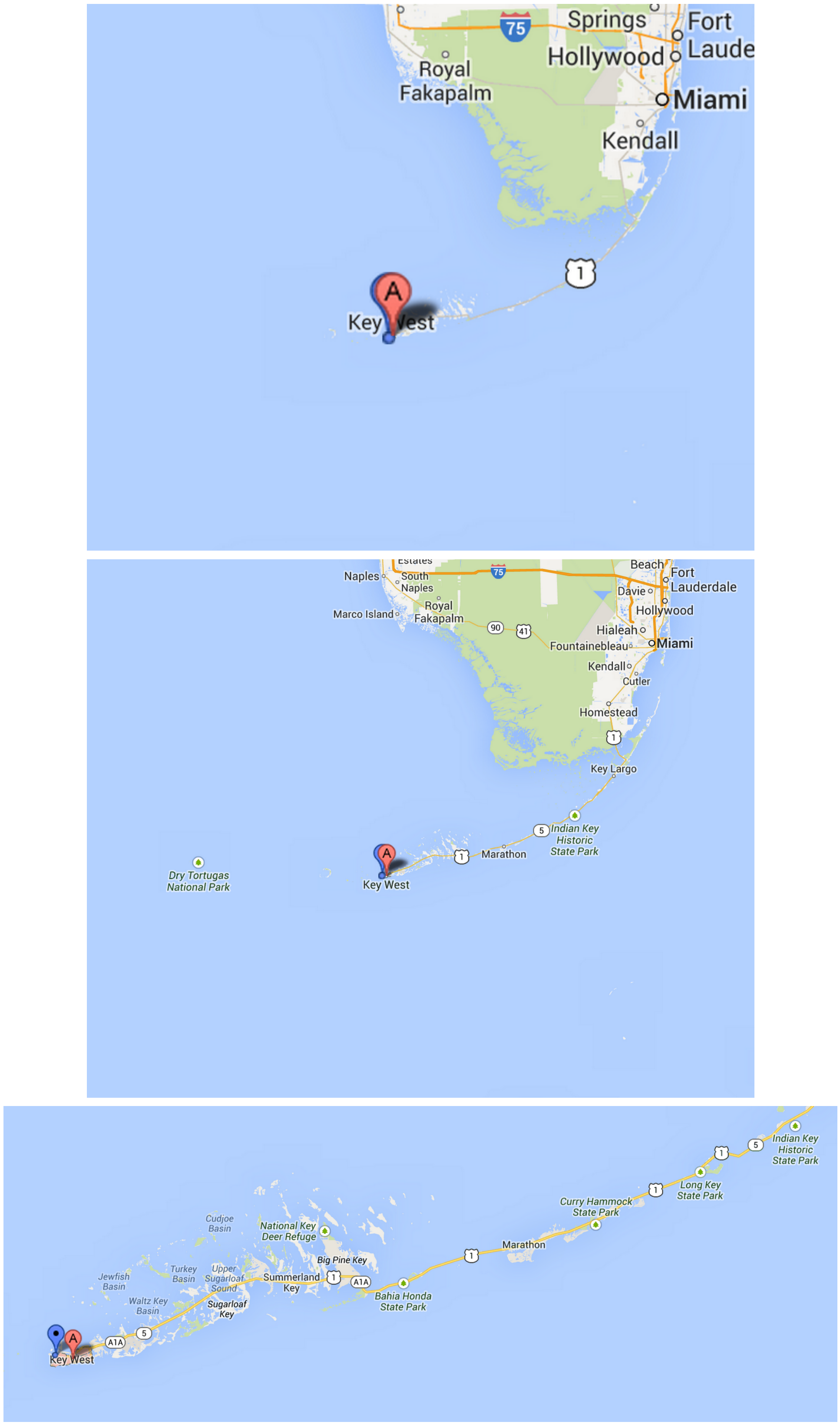
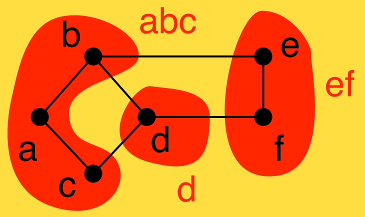


Figure 14.1: An illustration of graph contraction using maps. The road to key west at three different zoom levels. At first (top) there is just a road (an edge). We then see that there is an island (Marathon) and in fact two highways (1 and 5). Finally, we see more of the islands and the roads in between.

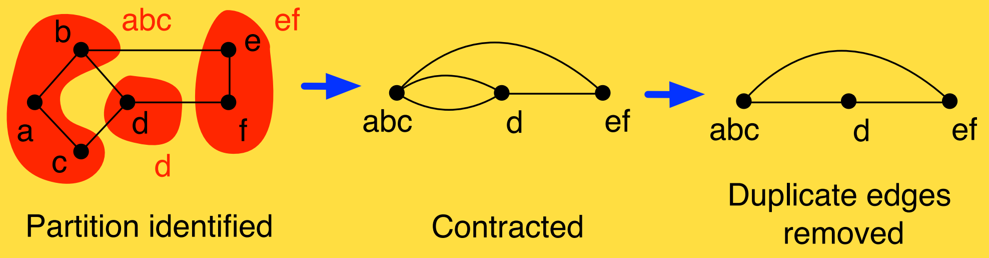
Example 14.23. *Partitioning the graph in Example 14.1 might generate the partitioning* $\{\{a,b,c\},\{d\},\{e,f\}\}$ *as indicated by the following shaded regions:*



We name the supervertices `abc`, `d`, *and* `ef`. *Note that each of the three partitions is connected by edges within the partition. Partitioning 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.*

Once we have partitioned the graph we can contract each partition into a single vertex. We note, however, that we now have to do something with the edges since their endpoints are no longer the original vertices, but instead are the new supervertices. The internal edges within each partition can be thrown away. For the cross edges we can relabel the endpoints to the new names of the supervertices. Note, however, this can create duplicate edges (also called parallel edges), which can be removed.

Example 14.24. *For Example 14.23 contracting each partition and replacing the edges.*



We are now ready to describe graph-contraction based on partitioning. To simplify things we assume that `partitionGraph` returns a new set of supervertices, one per partition, along with a table that maps each of the original vertices to the supervertex to which it belongs.

Example 14.25. *For the partitioning in Example 14.23,* `partitionGraph` *returns the pair:*

$$(\{\texttt{abc},\texttt{d},\texttt{ef}\},$$
$$\{\texttt{a}\mapsto\texttt{abc},\texttt{b}\mapsto\texttt{abc},\texttt{c}\mapsto\texttt{abc},\texttt{d}\mapsto\texttt{d},\texttt{e}\mapsto\texttt{ef},\texttt{f}\mapsto\texttt{ef}\})$$

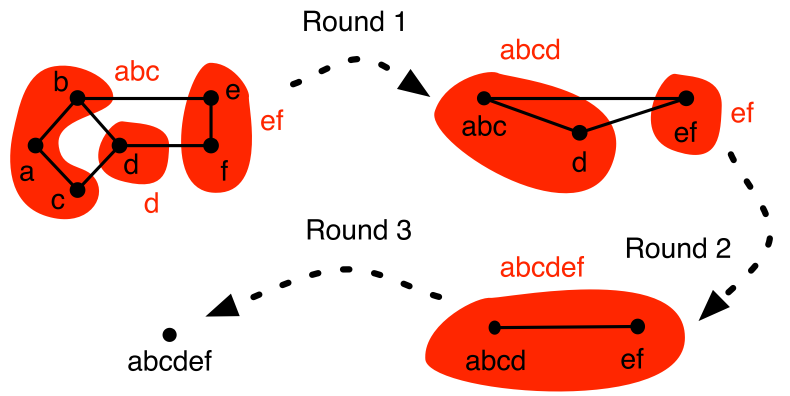


Figure 14.2: An example graph contraction. It completes when there are no more edges.

We can now write the algorithm for graph contraction as follows.

Algorithm 14.26 (Graph Contraction).

```
1 function contractGraph($G = (V, E)$) =
2 if $|E| = 0$ then $V$
3 else let
4       val $(V', P) =$ partitionGraph$(V, E)$
5       val $E' = \{(P[u], P[v]) : (u, v) \in E \mid P[u] \neq P[v]\}$
6   in
7       contractGraph$(V', E')$
8   end
```

This algorithm returns one vertex per connected component. It therefore allows us to count the number of connected components. An example is shown in Figure 14.2. As in the verbal description, `contractGraph` contracts the graph in each round. Each contraction on Line 4 returns the set of supervertices 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.

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

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

Computing the components. Our previous code just returns one vertex per component, and therefore allows us to count the number of components. It turns out we can modify the code slightly to compute the components themselves instead of returning their count. Recall than in the "contraction" based code for `scan` we did work both on the way down the recursion and on the way back up, when we added the results from the recursive call to the original elements to generate the odd indexed values. A similar idea will work here. The idea is to use the labels of the recursive call on the supervertices, to relabel all vertices. This is implemented by the following algorithm.

Algorithm 14.27 (Contraction-based graph connectivity)**.**

```
1 function  connectedComponents($G = (V, E)$) =
2 if  $|E| = 0$  then  $(V, \{v \mapsto v : v \in V\})$
3 else  let
4       val  $(V', P) =$ partitionGraph$(V, E)$
5       val  $E' = \{(P[u], P[v]) : (u, v) \in E \mid P[u] \neq P[v]\}$
6       val  $(V'', P') =$ connectedComponents$(V', E')$
7    in
8       $(V'', \{v \mapsto P'[P[v]] : v \in V\})$
9    end
```

In addition to returning the set of component labels as returned by *contractGraph*, this algorithm returns a mapping from each of the initial vertices to its component. Thus for our example graph it might return:

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

since there is a single component and all vertices will map to that component label.

The only difference of this code from *contractGraph* is that on the way back up the recursion instead of simply returning the result (V'') we also update the representatives for V based on the representatives from V''. 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 8 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 connected component (will be a label from V''). 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 base case of the *components* algorithm labels every vertex with itself.

Partitioning the Graph. In our discussion so far, we have not specified how to partition the graph. There different ways to do it, which corresponds to different forms of graph contraction.

Question 14.28. *What properties are desirable in forming the graph partitions to contract, i.e. the partitions generated by* `graphPartition`*.*

There are a handful of properties we would like when generating the partitions, beyond the requirement that each partition must be connected. Firstly we would like to be able to form the partitions without too much work. Secondly we would like to be able to form them in parallel. After all, one of the main goals of graph contraction is to parallelize various graph algorithms. Finally we would like the number of partitions to be significantly smaller than the number of vertices. Ideally it should be at least a constant fraction smaller. This will allow us to contract the graph in $O(\log |V|)$ rounds. Here we outline three methods for forming partitions that will be described in more detail in the following sections.

Edge Contraction: Each partition is either a single vertex or or two vertices with an edge between them. Each edge will contract into a single vertex.

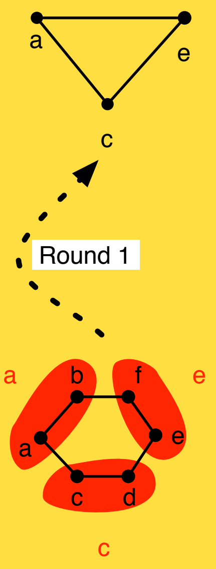
Star Contraction: Each partition is a *star*, which consists of a center vertex v and some number (possibly 0) satellite vertices connected by an edge to v. There can also be edges among the satellites.

Tree Contraction: For each partition we have a spanning tree.

14.2.1 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. We then contract the edges to a single vertex.

Example 14.29. *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.

Remark 14.30. *Finding a set of disjoint edges is also called* vertex matching*, since it tries to match every vertex with another vertex (monogamously), and is a standard problem in graph theory. In fact finding the vertex matching with the maximum number of edges is a well known problem. Here we are not concerned if it has a maximum number of edges.*

Question 14.31. *Can you describe an algorithm for finding a vertex matching?*

One way to find a vertex matching is to start at a vertex and pair in up with one of its neighbors. This can be continued until there are no longer any vertices to pair up, i.e., all edges are already incident on a selected pair. The pairs along with any unpaired vertices form the partition. The problem with this approach is that it is sequential.

Question 14.32. *Can you think of a way to find a vertex matching in parallel?*

An algorithm for forming the pairs in parallel will need to be able to make local decisions at each vertex. One possibility is in for each vertex in parallel to pick one of its neighbors to pair up with.

Question 14.33. *What is the problem with this approach?*

The problem with this approach 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 14.34. *Can you think of a way to use randomization to break this symmetry?*

We can use randomization to break the symmetry. One approach is to flip a coin for each edge and pick the edge if the edge (u, v) flips heads and all the edges incident on u and v flip tails.

Question 14.35. *Can you see how many vertices we would pair up in our cycle graph example?*

Lets analyze this approach on a graph that is a cycle. 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. The expectation of indicator random variables is the same as the probability it has value 1 (true). Since the coins are flipped independently at random, the probability that a vertex picks heads and its two neighboring edges pick tails is $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$. Therefore we have $E[R_e] = 1/8$.

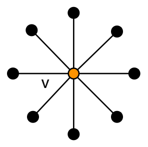
Thus summing over all edges, we conclude that expected number of edges deleted is $\frac{m}{8}$ (note, $m = n$). In the chapter on randomized algorithms Section 8.3 we argued that if each round of an algorithm shrinks the size by a constant fraction in expectation, and if the random choices in the rounds are independent, then the algorithm will finish in $O(\log n)$ rounds with high probability. Recall that all we needed to do is multiply the expected fraction that remain across rounds and then use Markov's inequality to show that after some $k \log n$ rounds the probability that the problem size is a least 1 is very small.

For a cycle graph, this technique leads to an algorithm for graph contraction with linear work and $O(\log^2 n)$ span.

Question 14.36. *Can you think of a way to improve the expected number of edges contracted?*

We can improve the probability that we remove an edge by letting each edge pick a random number in some range and then 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 in a cycle to $\frac{m}{3}$.

Question 14.37. *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 contraction. Thus in general, using edge contraction, we can shrink the graph only by a small amount.

As an example, consider a star graph:

Definition 14.38 (Star). *A* star *graph $G = (V, E)$ is an undirected graph with a center vertex $v \in V$, and a set of edges $E = \{\{v, u\} : u \in V \setminus \{v\}\}$.*

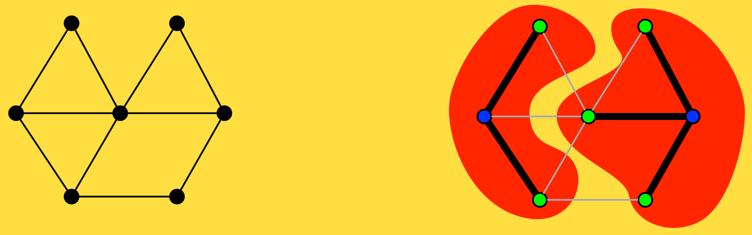
In words, a star graph is a graph made up of a single vertex v in the middle (called the center) and all the other vertices hanging off of it; these vertices are connected only to the center.

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. To fix this problem we need to be able to form partitions that are more than just edges.

14.3 Star Contraction

We now consider a more aggressive form of contraction. 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 14.39. *In the graph below (left), we can find 2 disjoint stars (right). The centers are colored blue and the neighbors are green.*



The question is how to identify the disjoint stars to form the partitioning.

Question 14.40. *How can we find disjoint stars?*

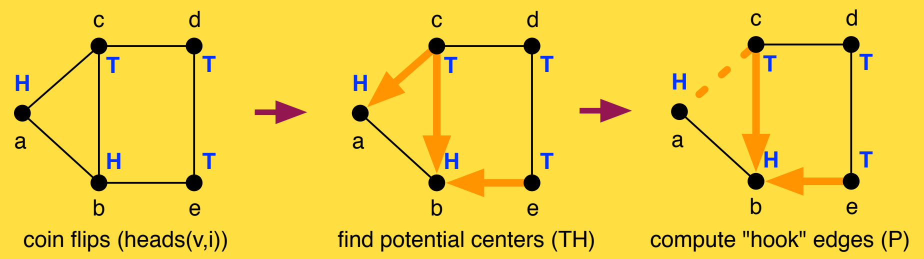
Similar to edge contraction, it is possible to to construct stars sequentially—pick an arbitrary vertex, attach all its neighbors to the star, remove the star from the graph, and repeat. However, again, we want a parallel algorithm that makes local decisions. As in edge contraction when making local decisions, we need a way to break the symmetry between two vertices that want to become the center of the star.

Question 14.41. *Can you think of a randomized approach for selecting stars (centers and satellites)?*

As with edge contraction we can use randomization to identify stars. An algorithm can use coin flips to decide which vertices will be centers and which ones will be satellites, and can then decide how to pair up each satellite with a center. To determine the centers, the algorithm can 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 is a potential satellite—it is only a potential satellite because quite possibly, none of its neighbors flipped a head so it has no center to hook 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 arbitrarily choose any center among its neighbors.

Example 14.42. *An example star partition. Coin flips turned up as indicated in the figure.*



Before describing the algorithm for partitioning a graph into stars, we need to say a couple words about the source of randomness. What we will assume is that each vertex is given a (potentially infinite) sequence of random and independent coin flips. The i^{th} element of the sequence can be accessed

$$\mathtt{heads}(v,i) : \mathtt{vertex} \times \mathtt{int} \to \mathtt{bool}.$$

Algorithm 14.43 (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   function starPartition$(G = (V, E), i) =$
5   let
6       % select edges that go from a tail to a head
7       val $TH = \{(u, v) \in E \mid \neg heads(u, i) \wedge heads(v, i)\}$
8       % make mapping from tails to heads, removing duplicates
9       val $P = \cup_{(u,v) \in TH} \{u \mapsto v\}$
10      % remove vertices that have been remapped
11      val $V' = V \setminus domain(P)$
12      % Map remaining vertices to themselves
13      val $P' = \{u \mapsto u : u \in V'\} \cup P$
14  in $(V', P')$ end
```

The function returns true if the i^{th} flip on vertex v is heads and false otherwise. 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. The algorithm for star contraction is given in Algorithm 14.43.

In our example graph, the function $heads(v, i)$ 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 13, 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 $domain(P) = \{c, e\}$ 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 14.44. *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 $v \in$ `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

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

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

Exercise 14.45. *What is the probability that a vertex with degree d is removed.*

Cost Specification 14.46 (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.*

14.3.1 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 $\mathbf{E}\left[X_n\right] = n/4$ 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 $\mathbf{E}\left[W(n, m)\right] = O(n + m \log n)$. Altogether, this gives us the following theorem:

Theorem 14.47. *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.*

14.3.2 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.