Chapter 13

Graph Contraction and Connectivity

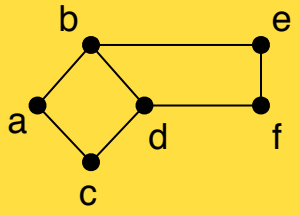
So far we have mostly covered techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of them are easy to parallelize while others are not. For example, we saw that BFS has some parallelism since each level can be explored in parallel, but there was 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 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 that restructure graphs.

13.1 Preliminaries

We start by reviewing and defining some graph terminology. In this chapter we will only concern ourselves with undirected graphs.

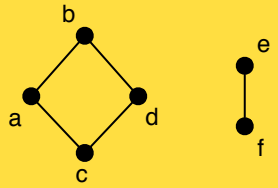
Example 13.1. *We will use the following undirected graph as an example.*



[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 13.2. *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. 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 13.3 (Subgraph)**.** *Let* $G = (V, E)$ *and* $H = (V', E')$ *be two graphs.* H *is a subgraph of if* $V' \subseteq V$ *and* $E' \subseteq E$.

Note that since H is a graph, it must be the case that both endpoints of all of its edges are incident on its vertices (i.e. for an undirected graph $E' \subseteq \binom{V'}{2}$). There are many possible subgraphs of a graph.

Exercise 13.4. *How many subgraphs are there of a triangle consisting of three vertices and three edges connecting them?*

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

Definition 13.5 ((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 to H without disconnecting H. In general when an object is said to be a *maximal* "X", it means that nothing more can be added to the object without violating the property "X". 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:

Problem 13.6 (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 13.7 (Vertex-Induced Subgraph)**.** *The subgraph of $G = (V, E)$ induced by $V' \subseteq V$ is the graph $H = (V', E')$ where $E' = \{\{u, v\} \in E \mid u \in V', v \in V'\}$.*

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 13.8. *Connected components on the graph in Example 13.2 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 13.9. *In Example 13.1 the partitioning of the vertices $\{\{a, b, c\}, \{d\}, \{e, f\}\}$ 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.*

13.2 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. The graph connectivity problem can be solved using a technique we have already covered, graph search. 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

Algorithmic Approach 13.10 (Graph Contraction)**.**

Base case *: For a small enough graph (e.g. no edges remaining), calculate the desired result directly.*

Inductive case *:*

- *Contract the graph into a smaller graph, ideally a constant fraction smaller.*
- *Recurse on the smaller graph.*
- *Use the result from the recursion along with the initial graph to calculate the desired result.*

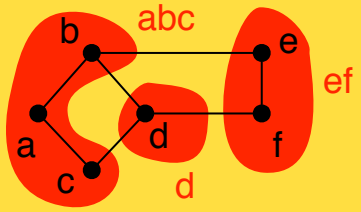
has not already been searched, search from it to create the second component. We repeat until all vertices have been checked. Either BFS or DFS can be used for the individual searches.

Using BFS or DFS lead to perfectly sensible sequential algorithms for graph connectivity, but they are not good parallel algorithms. Recall that DFS has linear span. 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). The diameter of a component can be as large as $n - 1$. A "chain" of n vertices will have diameter $n - 1$. 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.

We are instead interested in an approach that can identify all the components in parallel, and ideally lead to an algorithm for which the span is independent of the diameter. 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 use the same algorithmic technique we used for implementing the `scan` function in Chapter 5, *contraction*, although in this case we will contract the graph instead of a sequence. The key idea is to shrink the graph, ideally by a constant fraction in size, 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. 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. The graph contraction approach is summarized in Algorithmic Approach 13.10.

The question remains of how to contract the graph. If you recall, in the implementation of the `scan` we combined values. The idea for graphs is similar, but we combine vertices. In particular we partition the vertices of a graph into subsets and then contract each of the subsets (actually their induced subgraphs) into a single vertex, which we will refer to as a *supervertex*. To be useful for graph connectivity, however, we have to be careful how we partition the graph. We want to make sure the contraction maintains the connectivity of the graph—i.e., a component should be connected after a contraction if and only if it was connected before the contraction. For this purpose we will only select partitions for which each induced subgraph is connected.

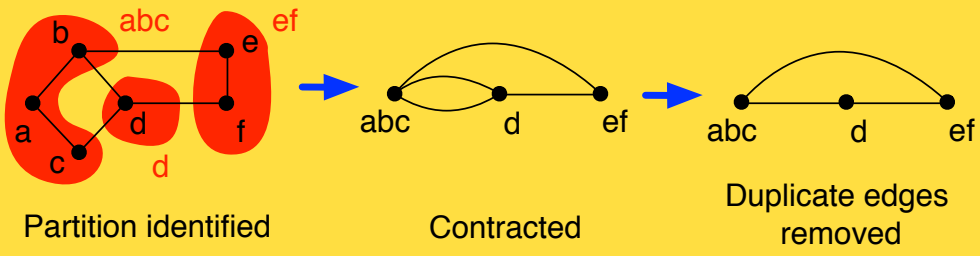
Example 13.11. *Partitioning the graph in Example 13.1 might generate the partitioning* $\{\{a, b, c\}, \{d\}, \{e, f\}\}$ *as indicated by the following shaded regions:*



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.*

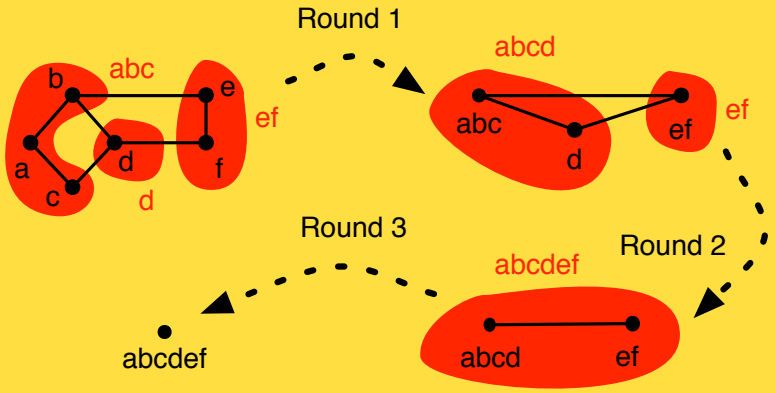
Once we have partitioned the graph we can contract each partition into a single vertex. We note, however, that we now have to do something with the edges since their endpoints are no longer the original vertices, but instead are the new supervertices. The internal edges within each partition can be thrown 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 13.12. *For Example 13.11 contracting each partition and replacing the edges.*



In graph contraction this step is applied recursively until we are left with no edges.

Example 13.13. *Contracting a graph down to a single vertex in three rounds.*



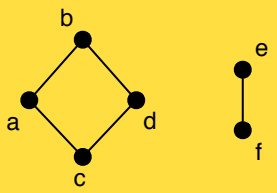
We are now almost ready to describe an algorithm that uses graph contraction, although we have not yet described how to find the partitioning and we need to be concrete about how to represent the graph and the partitioning. For now we will delay the question of how to find the partition (see Section 13.3) and instead assume that we are given a function `partitionGraph` which given a graph 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 13.14. *For the partitioning in Example 13.11, `partitionGraph` returns the pair:*

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

For the graph we will assume it is represented as an edge set. To simplify things, instead of assuming every edge is itself a set, we assume it an ordered pair, but appears in both orders. This is effectively equivalent to a directed graph in which we always have arcs in both directions.

Example 13.15. *The representation of an undirected graph as a set of ordered pairs, with each edge appearing in both directions.*



$$\begin{aligned} V &= \{a, b, c, d, e, f\} \\ E &= \{(a, b), (b, a), (b, d), (d, b), (a, c), (c, a), (c, d), (d, c), (e, f), (f, e)\} \end{aligned}$$

Algorithm 13.16 (Counting Components using Graph Contraction)**.**

```
1 function countComponents(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) ∈ E | P[u] ≠ P[v]}
6    in
7       countComponents(V', E')
8    end
```

We can now write an algorithm based on graph contraction that counts the number of connected components in a graph—see Algorithm 13.16. 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.

Example 13.17. *The values of V', P, and E' after each round of the contraction shown in Example 13.13.*

$$
\begin{array}{rcl}
 & V' & = \{\texttt{abc}, \texttt{d}, \texttt{ef}\} \\
\textit{round } 1 & P' & = \{\texttt{a} \mapsto \texttt{abc}, \texttt{b} \mapsto \texttt{abc}, \texttt{c} \mapsto \texttt{abc}, \texttt{d} \mapsto \texttt{d}, \texttt{e} \mapsto \texttt{ef}, \texttt{f} \mapsto \texttt{ef}\} \\
 & E' & = \{(\texttt{abc}, \texttt{ef}), (\texttt{ef}, \texttt{abc}), (\texttt{abc}, \texttt{d}), (\texttt{d}, \texttt{abc}), (\texttt{d}, \texttt{ef}), (\texttt{ef}, \texttt{d})\} \\
 & V' & = \{\texttt{abcd}, \texttt{ef}\} \\
\textit{round } 2 & P' & = \{\texttt{abc} \mapsto \texttt{abcd}, \texttt{d} \mapsto \texttt{abcd}, \texttt{ef} \mapsto \texttt{ef}\} \\
 & E' & = \{(\texttt{abcd}, \texttt{ef}), (\texttt{ef}, \texttt{abcd})\} \\
 & V' & = \{\texttt{abcdef}\} \\
\textit{round } 3 & P' & = \{\texttt{abcd} \mapsto \texttt{abcdef}, \texttt{ef} \mapsto \texttt{abcdef}\} \\
 & E' & = \{\}
\end{array}
$$

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 $\{\{\texttt{a}, \texttt{b}, \texttt{c}\}, \{\texttt{d}\}, \{\texttt{e}, \texttt{f}\}\}$ as the pair

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

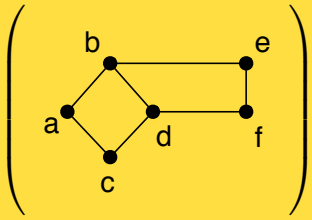
From now on we will name supervertices with a representative.

Algorithm 13.18 (Contraction-based graph connectivity)**.**

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

Computing the components. Our previous algorithm just counted the number of components. It turns out we can modify the algorithm 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 Algorithm 13.18. This algorithm returns the label of each component, along with a mapping from each of the initial vertices to its component label.

Example 13.19.

`connectedComponents` 

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. In this case `a` *was picked as the representative, but any of the initial vertices is a valid representative, in which case all vertices would map to it.*

The changes from `countComponents` is the that `connectedComponents` does something different in the base case and does some work when returning from the recursive call (Line 8). The base instead of returning the size of V returns all vertices in V along with a mapping from each one to itself. This is a valid answer since if there are no edges each vertex is its own component. In the inductive case, when returning from the recursion, Line 8 updates the mapping P from vertices to supervertices by looking up the component that the supervertex belongs to, which is given by P'. This simply involves the look up $P'[s]$ for every $(v \mapsto s) \in P$.

Example 13.20. *Consider our example graph (Example 13.19), and assume that* `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 $\texttt{connectedComponents}(V', E')$ *will map all vertices in* V' *to the same vertex. Lets say this vertex is* `a` *giving:*

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

Now $\{v \mapsto P'[s] : (v \mapsto s) \in P\}$ *will for each vertex-supervertex pair in* P*, look up what that supervertex got mapped to in the recursive call. For example, vertex* `f` *maps to vertex* `e` *in* P *so we look up* `e` *in* P'*, which gives us* `a` *so we know that* `f` *is in the component* `a`*. Overall the result is:*

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

13.3 Partitioning the Graph

We have postponed describing how to implement `partitionGraph`, which identified subsets of vertices (and their induced subgraphs) to be contracted into a supervertices. Recall that a property of the graph partitioning is that each partition must be connected. This was important so that connectivity is maintained on each round. Beyond maintaining connectivity, which is critical for correctness, there are other properties we would like when generating the partitions that are important for efficiency. 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. In this section we consider two simple methods for forming partitions, each which leads to a different form of graph contraction.

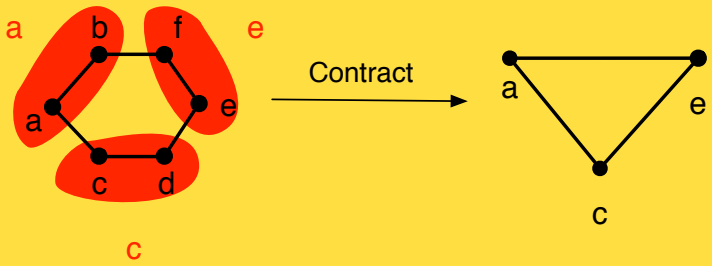
Edge Contraction: Each partition is either a single vertex or two vertices with an edge between them.

Star Contraction: Each partition is identified by a star, which consist of a center vertex and any number of neighboring satellite vertices. The satellites can have edges between them.

Edge Contraction

In edge contraction, the idea is to generate a set of partitions each consisting of either two vertices connected by an edge, or a single vertex.

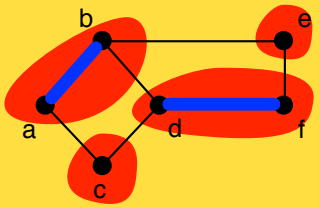
Example 13.21. *An example edge contraction in which every partition consists of two vertices and an edge between them. This partitioning will reduce the graph to half its size after contraction.*



Note that in general we cannot just have pairs of vertices since the graph might not have an even number of vertices, but even if it does (no pun intended), it is likely that it cannot be partitioned into a set of pairs joined by edges. We therefore will be satisfied by some set of disjoint edges (edges that do not share an endpoint). Finding such a set is a common task in a variety of graph algorithms, and hence has a name.

Definition 13.22. *A* vertex matching *for an undirected graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that no two edges in M share an endpoint.*

Example 13.23. *A vertex matching for our favorite graph (highlighted edges).*



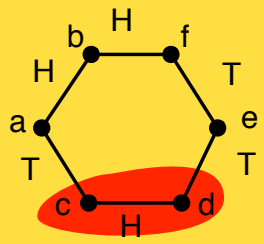
It defines four partitions (circled), two of them defined by the edges in the matching, $\{\texttt{a}, \texttt{b}\}$ and $\{\texttt{d}, \texttt{f}\}$, and two of them are the left over vertices $\texttt{c}$ and $\texttt{e}$.

Remark 13.24. *The problem of finding the largest vertex matching for a graph is called the* maximum vertex matching *problem. It is a well studied problems and there are several interesting algorithms for the problem, including one that can solve the problem in $O(\sqrt{|V|}|E|)$ work. In this section we do not care whether the matching is maximum, only that it is large enough, and would like to do it with only linear work.*

One way to find a vertex matching is to go through the edges one by one maintaining an initially empty set M and for each edge, if no edge in M is already incident on its endpoints then add it to M, otherwise toss it. The problem with this approach is that it is sequential since each decision depends on previous decisions. If we want to find the vertex matching in parallel we will likely need 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. The problem with this approach is that multiple vertices might pick edges that connect to the same other vertex. We therefore need a way to *break the symmetry* that arises when two vertices try to pair up with the same vertex.

It turns out that we can use randomization to break the symmetry. One approach for identifying a vertex matching in parallel is to flip a coin for each edge and pick the edge if it flips heads and all the edges incident on its endpoints flip tails. This guarantees that no two edges incident on the same vertex are selected. Let us analyze how effective this approach is in selecting a reasonably large set of edges. We first consider cycle graphs, consisting of a single cycle and no other edges. In such a graph every vertex has exactly two neighbors.

Example 13.25. *A graph consisting of a single cycle.*



Each edge flips a coin that comes up either heads (H) or tails (T). We pick an edge if it turns up heads and all other edges incident on its endpoints are tails. In the example only the edge $\{c, d\}$ is selected.

We want to determine the probability that an edge is selected in such a graph. Since the coins are flipped independently at random, and the vertices at each endpoint each have one other neighbor, the probability that an edge picks heads and its two neighboring edges pick tails is $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$. We now want to analyze how many edges are selected in expectation. 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. Recall that the expectation of indicator random variables is the same as the probability it has value 1 (true). 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 a cycle graph). In the chapter on randomized algorithms Section 7.3 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

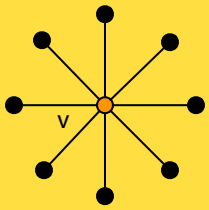
leads to an algorithm for graph contraction with linear work and $O(\log^2 n)$ span—left as an exercise.

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 $\frac{m}{3}$, which is significantly better than $\frac{m}{8}$.

Although edge contraction works quite well with cycle graphs, we should ask if it works well with arbitrary graphs? Unfortunately it does not. 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. 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. For example, consider the following kind of graph.

Definition 13.26 (Star Graph)**.** *A* star *graph $G = (V, E)$ is an undirected graph with a* center *vertex $v \in V$, and a set of edges E that attach v directly to the rest of the vertices, called* satellites*, i.e. $E = \{\{v, u\} : u \in V \setminus \{v\}\}$.*

Example 13.27. *A star graph with center v and eight satellites.*

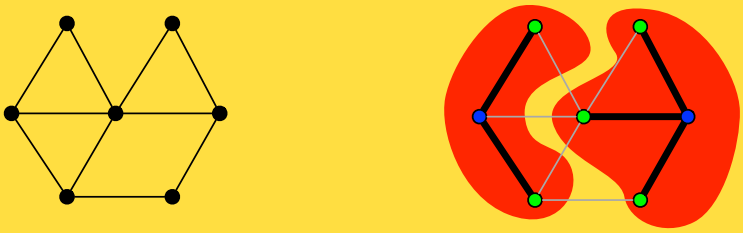


It is not difficult to convince ourselves that on a star graph with n vertices—1 center and $n - 1$ satellites—any edge contraction algorithm will take $\Omega(n)$ rounds. To fix this problem we need to be able to form partitions that consist of more than just edges.

Star Contraction

We now consider a more aggressive form of contraction. The idea is to partition the graph into a set of graphs that each contain a star graph.

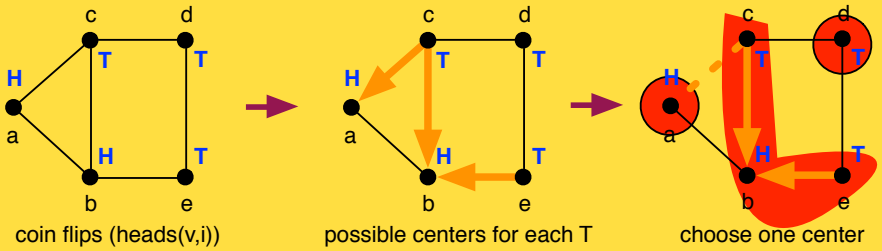
Example 13.28. *In the graph below (left), we can find 2 disjoint stars (right). The centers are colored blue and the neighbors (satellites) are green.*



The question is how to identify the disjoint stars to form the partitioning. As with edge contraction, it is possible 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. Again we can use randomization to identify stars. To determine the centers, the algorithm can flip a coin on each vertex. If the coin 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 join.

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 potential satellite arbitrarily choose any center among its neighbors, if it has any neighboring center.

Example 13.29. *An example star partition. Coin flips turned up as indicated in the figure, where H indicates a center, and T indicates a potential satellite.*



The vertex `c` *neighbors both* `a` *and* `b` *which are both centers. In the example it chooses* `b` *to join. The vertex* `d` *has no neighboring centers, so it is left on its own. We end up forming three partitions—the star with center* `a` *(with no satellites), the star with center* `b` *(with two satellites), and the singleton* `d`*.*

Algorithm 13.30 (Star Partition)**.**

```
1  function starPartition(G = (V, E), i) =
2  let
3      % select edges that go from a satellite to a center
4      val TH = {(u, v) ∈ E | ¬heads(u, i) ∧ heads(v, i)}
5      % Use table merge to make a mapping from satellites to centers, removing duplicates
6      val P = ∪_{(u,v)∈TH} {u ↦ v}
7      % The supervertices − centers and unmatched satellites
8      val V' = V \ domain(P)
9      % Map supervertices to themselves
10     val P' = {u ↦ u : u ∈ V'}
11 in (V', P' ∪ P) end
```

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

$$\texttt{heads}(v, i) : V \times \mathbb{Z} \to \mathbb{B} \ .$$

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 13.30. It takes in a graph and a round number, and returns graph partitioning of the sort returned by `partitionGraph`. The algorithm flips coins on each vertex and selects those that point from tail (satellite) to head (center). It then merges all these edges using table merge. This effectively decides for each satellite one of the centers it will point to. All the centers and any potential satellites that do not get mapped to a center, since they do not neighbor a center, are mapped to themselves (Line 10). Finally we merge the table for the remapped satellites and the other vertices.

Example 13.31. *Returning to Example 13.29 and assuming the same flips as given in that example, we have that:*

$$TH = \{(c, a), (c, b), (e, b)\} \ .$$

These are the (directed) edges from satellites to centers. Now we convert each edge into a singleton map, and merge them into the mapping:

$$P = \{c \mapsto b, e \mapsto b\} \ .$$

Note that the edge (c, a) has been removed since the merging of the map selects only one element for each key in the domain. Now for all remaining vertices $V' = V \setminus \texttt{domain}(P) = \{a, b, d\}$ we map them to themselves, giving:

$$P' = \{a \mapsto a, b \mapsto b, d \mapsto d\} \ .$$

The P and P' are merge to give the final mapping:

$$P' \cup P = \{a \mapsto a, b \mapsto b, c \mapsto b, d \mapsto d, e \mapsto b\} \ .$$

Analysis of Star Contraction. When the stars found by `starPartition` are contracted, 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 $n/4$ in expectation.

Lemma 13.32. *For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by $\texttt{starPartition}(G, r)$. 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 13.33. *What is the probability that a vertex with degree d is removed.*

Cost Specification 13.34 (Star Contraction). *Using* `ArraySequence` *and* `STArraySequence`*, we can implement* `starPartition` *reasonably efficiently in* $O(n+m)$ *work and* $O(\log n)$ *span for a graph with* n *vertices and* m *edges.*

13.4 Returning to Connectivity

Now lets analyze the cost of `countComponents` and `connectedComponents` when using `starPartition`. 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 `starPartition`). But $\mathbf{E}[X_n] = n/4$ so $\mathbf{E}[n'] = 3n/4$. This is a familiar recurrence, which we know solves to $O(\log^2 n)$.

As for work, ideally, we would like to show that the overall work is linear since we might hope that the size is going down by a constant fraction on each round. Unfortunately, this is not the case. Although we have shown that one can remove a constant fraction of the non-isolated vertices on one star contract round, we have not shown anything about how many edges we remove. We can argue that the number of edges removed is at least equal to the number of vertices since removing a satellite 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 13.35. *For a graph* $G = (V, E)$*,* `countComponents` *using* `starPartition` *graph contraction with an array sequence works in* $O(|V| + |E| \log |V|)$ *work and* $O(\log^2 |V|)$ *span.*

Contracting Trees

Our analysis in the previous section was for general graphs. What if we are contracting a forest of trees instead. 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 star is a special case of a tree, since it has no cycles.

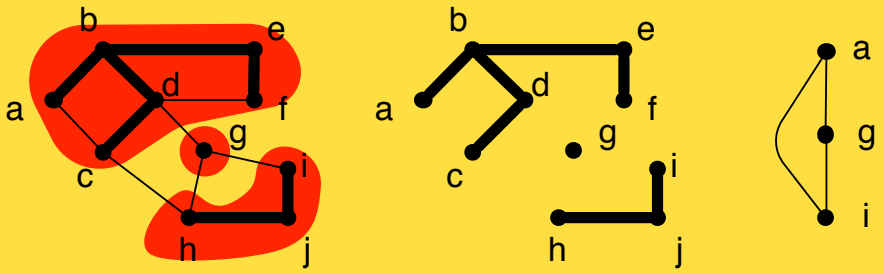
The same `connectedComponents` algorithm based on star contraction can be used for a forest or tree as for general graphs, but if we use it on trees the assymptotic bound on the work is better. This is because the number of edges is a forest is never more than the number of vertices, and hence the number of edges must go down geometrically from step to step (in expectation), as do the number of vertices. The overal expected work will be a geometric sum of the form:

$$\mathbf{E}\left[W(n,m)\right] = \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i kn = O(n)\ ,$$

instead of $O(m \log n)$ for general graphs. The span is not affected.

For a graph $G = (V, E)$ consider a subset of edges $T \subset E$ that forms a forest (i.e. has no cycle). Such a subset defines a partitioning of the orginal graph, where each tree is its own partition. Therefore one way to contract a graph is to identify such a subset T, and then use `connectedComponents`(V, T), which does linear work as explained above, as our `partitionGraph` routine. We will use this idea in an algorithm for Minimum Spanning Trees described in the next chapter.

Example 13.36. *A graph and a subset of the edges T (in bold) that define a set of three disjoint trees, each implying a partition:*



If we run `connectedComponents` on T (the middle diagram) we are left with the desired partitioning with supervertices $\{a, g, i\}$ and the mapping:

$$\{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto a, e \mapsto a, f \mapsto a, g \mapsto g, h \mapsto i, i \mapsto u, j \mapsto i\}$$

This can be used to contract the original graph what is shown on the right.

.