Chapter 17

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.

17.1 Preliminaries and Graph Partitioning

Please review the material on graphs (Section 2.4) before proceeding with the rest of this chapter, especially the sections on subgraphs and connectivity.

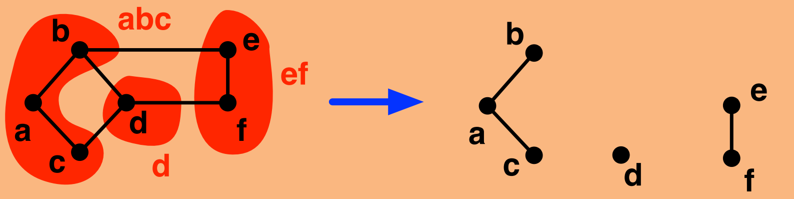
Recall that a partitioning of a set S is a set T of subsets of S such that each element of S is in exactly one subset $t \in T$. We refer to each element of T as a ***partition*** and the set T as a ***partitioning*** of S.

A ***vertex partitioning of a graph*** is a partitioning of its vertex set. In the context of graph partitioning, we refer to each subgraph induced by a subset of the vertices as a partition. We can also define edge partitioning of a graph similarly. In this chapter, we are only concerned with vertex partitioning of graphs; we therefore use the term "graph partitioning" to refer to "vertex-partitioning of a graph". In a graph partitioning, we can distinguish between two kinds of edges: internal edges and cross edges. ***Internal edges*** are edges that are within a partition; ***cross edges*** are edges that are between partitions. One way to partition a graph is to make each connected component a partition. In such a partitioning, there are no cross edges between the partitions.

[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.

Sometimes it is useful to give a name or a label to each partition. A partitioning of a graph can then be described as a set of names for the partitions and ***partition map*** that maps each vertex to the name of its partition. The names can be chosen arbitrarily, though sometimes it is conceptually and computationally easier to use a vertex inside a partition as a name (representative) for that partition.

Example 17.1. *The partitioning of the vertices* $\{\{a, b, c\}, \{d\}, \{e, f\}\}$ *defines three partitions as vertex-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.*
Having named the three partitions "abc*", "*d*" and "*ef*", we can specify this partitioning with following partition map:*
$(\{abc, d, ef\}, \; \{a \mapsto abc, b \mapsto abc, c \mapsto abc, d \mapsto d, e \mapsto ef, f \mapsto ef\})$

In our example, we gave fresh names to supervertices. It is often more convenient to pick a representative from each partition instead of a fresh name. We can then represent the partitioning as a mapping from each vertex to its representative (supervertex). For example, we can represent the partitioning in the example with the following pair consisting of the names for set of partitions and the partition map

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

17.2 Graph Contraction

Graph contraction is a technique for computing properties of graph in parallel. As its name implies, it is a contraction technique, where we solve a problem by reducing to a smaller instance of the same problem.

Question 17.2. *Can we solve graph problems using divide-and-conquer?*

Graph contraction is especially important because divide-and-conquer is difficult to apply to graphs efficiently. The difficulty is that divide-and-conquer would require reliably partitioning the graph into smaller graphs. Due to their irregular structure, graphs are difficult to partition. In fact, graph partitioning problems are typically NP-hard.

In the rest of this section, we describe the graph-contraction design technique and describe two approaches to contracting graphs by using edge partitionings and star partitionings, briefly referred to as ***edge contraction*** and ***star contraction***.

17.2.1 The Design Technique

As with the contraction technique that we have seen in Chapter 6, graph-contraction can be used to solve a variety of graph problems. The key idea behind graph contraction is to "shrink" the input graph, ideally by a constant factor in size so that we can solve the problem on a smaller graph, and then use the solution to the smaller graph to construct the solution for the input graph. This technique can be described as an inductive algorithm-design technique (Design Technique 17.3).

Algorithm-Design Technique 17.3 (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.*

The key step of graph contraction is the contraction of the graph into a smaller graph.

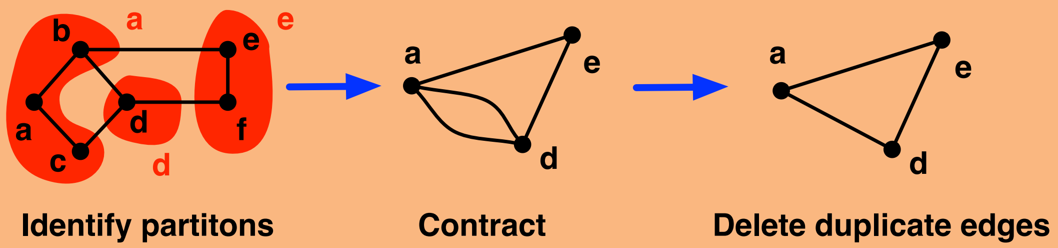
Question 17.4. *Any ideas about how we might contract a graph?*

One way to contract a graph is to map it to a smaller graph by mapping subsets of vertices, as defined by a partitioning, to vertices in a smaller graph as follows.

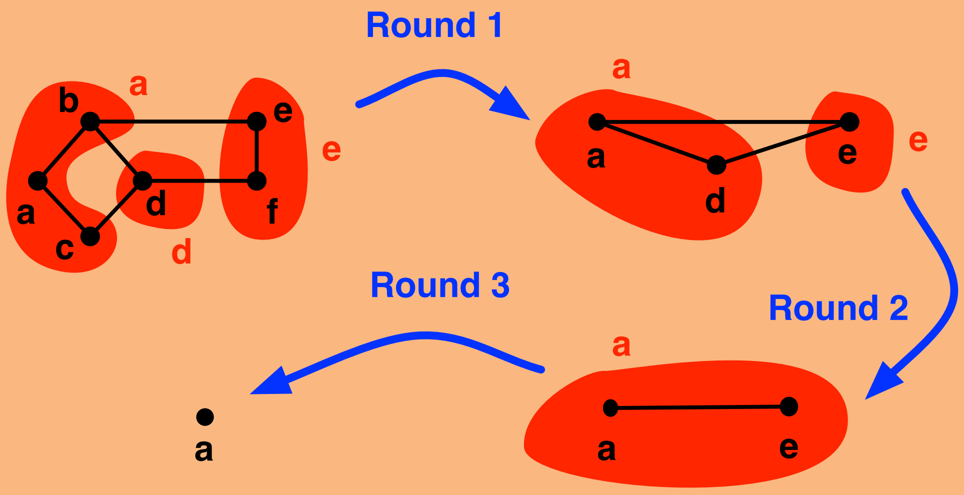
1. Compute a vertex-partitioning for the graph.
2. Contract each partition into a single vertex, called a *supervertex*.
3. Drop edges internal to a partition.
4. Reroute cross edges to corresponding supervertices.

By selecting a partitioning to guide the contraction, we make sure that each vertex in the graph is mapped to one unique vertex in the smaller graph (because partitions are disjoint and they include all the vertices). Given a partitioning, we contract the graph by contracting each partition into a single vertex and updating the edges. More specifically, we create a super vertex for each partition. We then consider each edge in the graph. If the edge is internal to a partition, we skip it. If it is a cross edge, we create a new edge between the supervertices representing the partitions of the vertices of the edge. One final point that we have to be careful is duplicate edges: since there can be many cross edges between two partitions, we may end up creating multiple edges between two supervertices. We can remove such edges or leave them in. This process of identifying a partition and updating the edges is called a round of graph contraction. In a graph contraction, rounds are repeated until there are no edges left.

Example 17.5. *One round of graph contraction illustrated.*



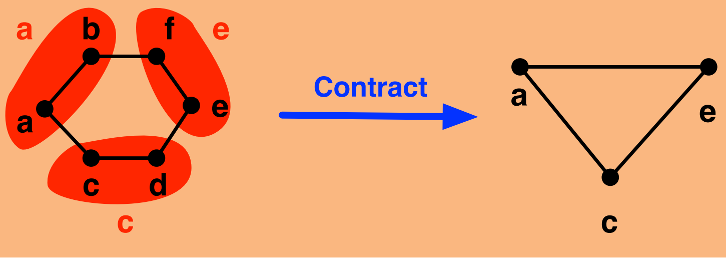
Contracting a graph down to a single vertex in three rounds:



17.2.2 Edge Partitioning and Edge Contraction

In edge partitoning, we select each partition to be either a single vertex or two vertices connected by an edge. A graph contraction where partitions are selected by edge partitioning is referred to as ***edge contraction***.

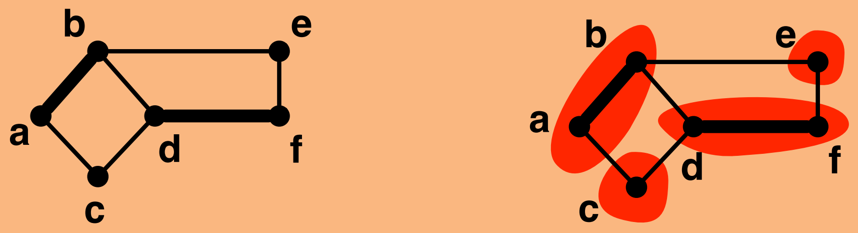
Example 17.6. *An example edge partitioning in which every partition consists of two vertices and an edge between them. This partitioning will reduce the graph to half its size after contraction.*



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 17.7. *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 17.8. *A vertex matching for our favorite graph (highlighted edges) and the corresponding partitions.*



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

Remark 17.9. *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. For graph contraction, we do not need a maximum matching but one that it is sufficiently large.*

Question 17.10. *Can you describe an algorithm for finding a vertex matching?*

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.

Question 17.11. *Can you think of a way to find a vertex matching in parallel?*

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.

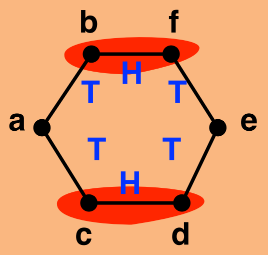
Question 17.12. *What is the problem with this approach?*

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.

Question 17.13. *Can you think of a way to use randomization to break this symmetry?*

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 17.14. *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 the edges $\{c, d\}$ and $\{b, f\}$ are 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).

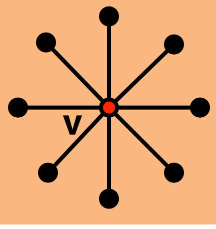
In the chapter on randomized algorithms Section 9.3 we argued that if each round of an algorithm shrinks the size by a constant fraction in expectation, and if the random choices in the rounds are independent, then the algorithm will finish in $O(\log n)$ rounds with high probability. Recall that all 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.

Question 17.15. *Can you think of a way to improve the expected number of edges deleted?*

There are several ways to improve the number of deleted edges. One way is for each edge to pick one of its neighbors and to select an edge (u, v) if it was picked by both u and v. In the case of a circle, this increases the expected number of deleted edges to $\frac{m}{4}$. Another way is let 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 circle to $\frac{m}{3}$.

Although edge partitioning works quite well with cycle graphs, or sequentially with the appropriate data structures, we should ask if it works well in parallel with arbitrary graphs? Unfortunately it does not. The problem is that only one edge incident on a vertex can be contracted on each round. Therefore vertices with high degrees, will have to contract their neighbors one at a time. For example, consider the following kind of graph, called a star graph.

Example 17.16. *A star graph with center v and eight satellites.*



More precisely, we can define a star graph as follows.

Definition 17.17 (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. $E = \{\{v, u\} : u \in V \setminus \{v\}\}$.*

Note that a single vertex and a single edge are both star graphs.

It is not difficult to convince ourselves that on a star graph with n vertices—1 center and $n - 1$ satellites—any edge partitioning 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.

Remark 17.18. *An abstract data type called disjoint sets is often used to contract graphs sequentially. Disjoint sets supply two functions:* union*, which joins two components, and* find*, which finds what component a vertex is in. In our framework, the union operation is simply edge contraction across a single edge, and the find is just a lookup in the partition map. Semantically for a partition map P we can define* `union` *as:*

$$\begin{aligned}\texttt{union}(P, u, v) \;=\; & \{u' \mapsto \textbf{if}\ (v' = u)\ \textbf{then}\ v\ \textbf{else}\ v' \\ & \ \ \textbf{:}\ (u' \mapsto v') \in P\}\end{aligned}$$

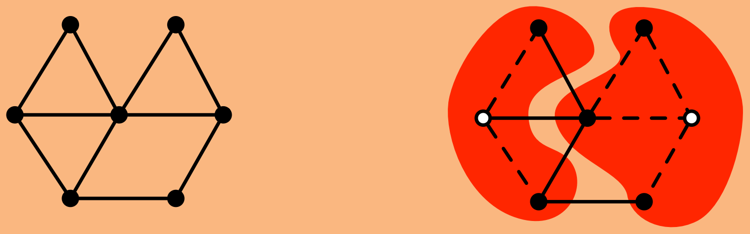
where here we have made v the new representative of the supervertex $\{u, v\}$, and have updated all vertices that used to point to u to now point to v. Implementing the `union` *this way, however, is inefficient since it can require updating a lot of vertices. It turns out that the operations can be can be implemented much more efficiently. Indeed one can implement a data structure that only requires amortized $O(\alpha(n))$ work per operation, where $\alpha(n)$ (the inverse Ackermann function) is a function that is very close ot $O(1)$, and n is the number of operations.*

17.2.3 Star Partitioning and Star Contraction

Since partitions must be disjoint, edge partitioning prevents contracting the graph significantly, because if we select an edge incident on a vertex v as a partition, then none of the other edges incident on that vertex can be their own partition. This is a fundamental limitation, because graphs can have high-degree vertices. Thus, to find a technique that works more broadly, we need to allow for high-degree vertices to be part of a partition.

We consider a more aggressive form of partitoning called star partitioning, where partitions are determined by selecting stars. For example in Example 17.16, the whole graph can be a single partition. We refer to a graph contraction where partitions are selected by start partitioning an ***star contraction***.

Example 17.19. *In the graph below (left), we can find 2 disjoint stars (right). The centers are highlighted (with color white), the rest of the vertices are satellites. Internal edges, which will reside inside a partition and will be removed during contraction, are drawn in dashes; cross edges remain solid. Note that the internal edges of each star is not always equal to the edges of the star.*



Question 17.20. *How can we star partition a graph?*

As with edge partitioning, it is possible to construct star partitionings sequentially. For example, we can start with an arbitrary vertex and attach all its neighbors to a star. We would then remove the star from the graph, and repeat (until all vertices are accounted for) by picking another arbitrary vertex. Note that a vertex is the simplest form of a star. So if we have isolated vertices remaining they form trivial stars.

Question 17.21. *How can we star partition a graph?*

We can also construct star partitionings in parallel by making local independent decisions in parallel. As in edge partitionings, we can use randomization to break symmetry.

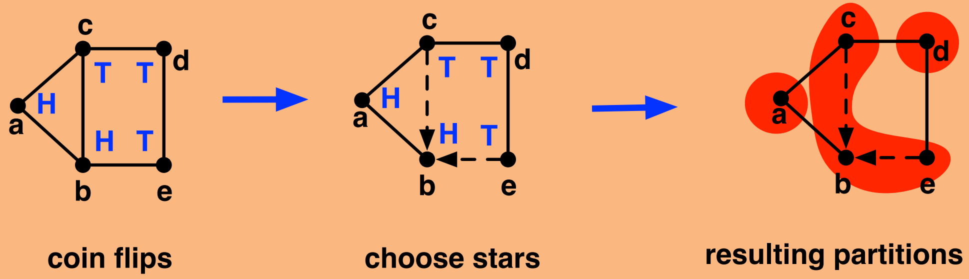
Question 17.22. *Can you think of a randomized approach for selecting stars?*

For example, to determine the centers, we can flip a coin for each vertex. If the coin comes up heads, that vertex is the center of a star. We can then make a vertex that has flipped tails a satellite by attaching it to one of its neighbors that is a center. If no such neighbor exists (all neighbors have flipped tails) then we make the vertex a center as well. If a vertex has multiple centers as neighbors, we can pick one arbitrarily. This approach may not be optimal in the sense that it may not create the smallest number of stars but this is acceptable for our purposes, because we are only interested in reducing the size of the contracted graph by a constant factor—not minimizing it.

Algorithm 17.24 (Star Partition)**.**

```
function starPartition$(G = (V, E), i)$ =
let
    % select edges that go from a satellite to a center
    val $TH = \{(u, v) \in E \mid \neg heads(u, i) \wedge heads(v, i)\}$

    % Use table merge to make a mapping from satellites to centers, removing duplicates
    val $P = \bigcup_{(u,v) \in TH} \{u \mapsto v\}$

    % The supervertices – centers and unmatched satellites
    val $V' = V \setminus domain(P)$

    % Map supervertices to themselves
    val $P' = \{u \mapsto u : u \in V'\}$

in $(V', P' \cup P)$ end
```

Example 17.23. *An example star partition. Vertices that flipped heads become centers (vertices* a, b*. A vertex (vertices* c *and* e*) that flipped tails attempts to become a satellite by finding a center among its neighbors, breaking ties arbitrarily. If none exist (example: vertex* d*), it becomes a center. 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*.*



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

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

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 partitioning is given in Algorithm 17.24. 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 the directed edges that point from tail (satellite) to head (center)—this gives TH. In this set of edges there might be multiple edges from the same satellite and we want to choose one of them. Line 6 does this by creating a set of singleton tables and merging them. In particular the union is shorthand for the following merge:

$$\texttt{Set.Reduce (Table.merge (}\textbf{fn}\texttt{ (x,y) => x)) } \emptyset \ \{\{u \mapsto v\} : (u,v) \in TH\}$$

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 17.25. *Returning to Example 17.23 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 Partitioning. 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 17.26. *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, $\textbf{Pr}\left[R_v\right] \geq \textbf{Pr}\left[T_v\right] \textbf{Pr}\left[H_u\right] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

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

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

Cost Specification 17.27 (Star Partitioning)**.** *Based on array-based cost specification for sequences and single-treaded sequences, the cost of* `starPartition` *is* $O(n+m)$ *work and* $O(\log n)$ *span for a graph with* n *vertices and* m *edges.*

17.3 Connectivity via Graph Contraction

An important, well-studied problem, called ***(graph) connectivity*** problem, is to determine the connected components of a graph.

Problem 17.28 (The Graph Connectivity (GC) Problem)**.** *Given an undirected graph* $G = (V, E)$ *return all of its connected components (maximal connected subgraphs).*

A graph connectivity algorithm would return the connected components of a graph, by for example specifying the set of vertices in each component.

Question 17.29. *Can you solve the graph-connectivity problem by using one of the techniques recently covered in this class?*

The graph connectivity problem can be solved by using graph search. In particular, we can start at any vertex and find, using DFS or BFS, all vertices reachable from it to create the first component. We then move onto the next vertex and if it has not already been searched, search from it to create the second component, and so on we repeat until we exhaustively consider all the vertices.

Question 17.30. *Would these approaches yield good parallelism? What would be the span of the algorithm?*

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

Question 17.31. *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 17.32. *How large can the diameter of a component be? Can you give an example?*

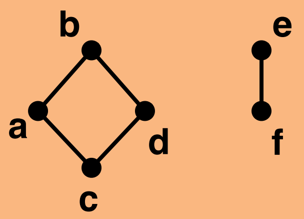
The diameter of a component can be as large as $n-1$. A "chain" of n vertices will have diameter $n-1$.

Question 17.33. *How about in cases when the diameter is small, for example when the graph is just a disconnected collection of edges.*

When the diameter of a graph is small, we may use BFS to perform each graph search, but we still have to iterate over the components one by one. Thus the span in the worst case can be linear in the number of components, which can be large.

We would like to find a parallel algorithm for connectivity that has a small span an all graphs. To this end, we will use the graph-contraction technique with star partitioning. To specify the algorithm, we will use an edge-set representation for graphs, where every edge is represented as a pair of vertices, in both orders. This is effectively equivalent to a directed graph representation of undirected graphs with two arcs per edge.

Example 17.34. *The representation of an undirected graph as a set of ordered pairs, with each edge appearing in both directions.*



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

We can now write given an algorithm based on graph contraction that counts the number of connected components in a graph (Algorithm 17.35. Each contraction on Line 4 returns the

Algorithm 17.35 (Counting Components using Graph Contraction)**.**

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

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 17.36. *The values of V', P, and E' after each round of the contraction shown in Example 17.5.*

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

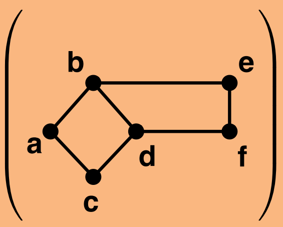
Our previous algorithm just counted the number of components. It turns out we can modify the algorithm slightly to compute the components themselves instead of returning their count. To this end, we are going to construct the mapping from vertices to their components recursively. This is possible because we can obtain the mapping by composing the mapping from vertices to their supervertices and the mapping from supervertices to their components, which we obtain recursively. Algorithm 17.37 shows the algorithm.

Algorithm 17.37 (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 $(V', P) =$ starPartition$(V, E)$
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         $(V'', \{v \mapsto P'[s] : (v \mapsto s) \in P\})$
9     end
```

Example 17.38.

`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 only differences from `countComponents` are a modification to the base case, and the extra line (Line 8) after the recursive call. In the base case 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 $(v \mapsto s) \in P$. Note that if you view a mapping as a function, then this is equivalent to function composition, i.e. $P' \circ P$.

Example 17.39. *Consider our example graph (Example 17.38), and assume that* `starPartition` *returns:*

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

Since the graph is connected, the recursive call to `connectedComponents`(V', E') *will map all vertices in* V' *to the same vertex. Lets say this vertex is* `a` *giving:*

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

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:*

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

Now lets analyze the cost of `countComponents` and `connectedComponents` when using `starPartition`. Let n_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):

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

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

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.

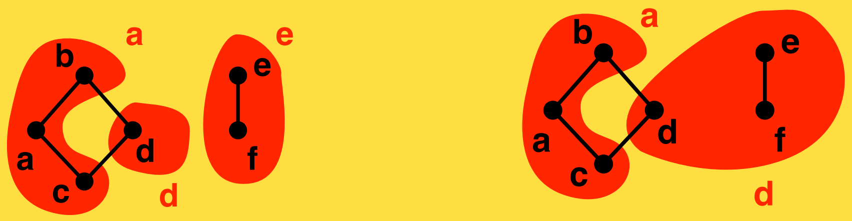
To bound the work, we will consider non-isolated and isolated vertices separately. For the non-isolated vertices, we have the following work recurrence:

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

where n'_n is the remaining number of non-isolated vertices as defined in the span recurrence. This solves to $\mathbf{E}\left[W(n_n, m)\right] = O(n_n + m \log n_n) = O(n + m \log n)$. To bound the work on isolated vertices, we note that there at most n of them at each round and thus, the additional work is $O(n \log n)$. This analysis gives us the following theorem.

Theorem 17.40. *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.*

Remark 17.41. *In general the graph contraction techniques does not specify how to perform the partitioning needed at each step. For example, in this chapter, we covered two techniques for partitioning. Depending on the problem, other techniques can be used as well. For graph contraction to be applicable to a problem, however, it is important that the graph contracted at each round satisfy certain properties. For example, when solving graph connectivity with the algorithms described here, we have to be careful that the partitioning maintains connectivity: a component should be connected after a round of graph contraction if and only if it was connected before the round. To ensure this, we will need to use a partitioning algorithm that ensures that each partition is connected. For example, of the partitionings of the graphs shown below, the second one does not maintain connectivity, whereas the first one does.*



The partitioning on the left is appropriate for graph contraction since each partition is connected. The partition on the right is not since `d` is not connected to `e` and `f`.

17.4 Tree Contraction

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. Since it has no cycles, a star is a tree.

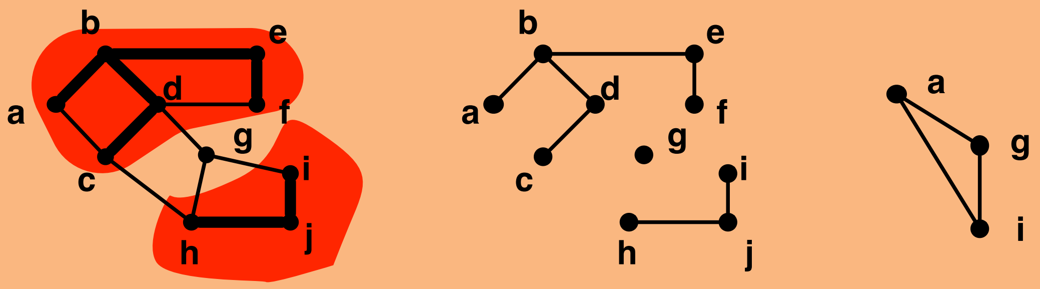
The same `connectedComponents` algorithm based on star partitoning can be used for a forest or tree but with trees, the work bound is better. This is because the number of edges is a forest is never more than the number of vertices, and hence the number of edges decrease geometrically (in expectation) in each round, as do the number of vertices. The overal expected work is therefore 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, instead of our `starPartition` routine. We will use this idea in an algorithm for Minimum Spanning Trees described in Chapter 18.

Example 17.42. *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.

17.5 Exercises and Problems

Exercise 17.43. *There are 18 subgraphs for a triangle consisting of three vertices and three edges connecting them, including the empty graph and the graph itself. List them all.*

Exercise 17.44. *In star contraction, what is the probability that a vertex with degree d is removed.*

Exercise 17.45. *Find an example graph, where star-based graph contraction removes a small number of edges on each round.*

.