Graph Contraction

So far we have mostly talking about standard techniques for solving problems on graphs that were developed in the context of sequential algorithms. Some of these are easy to parallelize while others are not. For example, we saw there was plenty of parallelism in the Bellman-Ford algorithm, and also in the all pairs shortest path algorithms since they are based on parallel application of Dijkstra (and perhaps Bellman Ford preprocessing if there are negative weights).

We are now going to discuss some techniques that will add to your toolbox for parallel algorithms. The first of these techniques is graph contraction. This is actually a reasonably simple technique and can be applied to a variety of problems including graph connectivity, spanning trees, and minimum spanning trees. In the discussion of graph contraction, we will assume that the graph is undirected unless otherwise stated. The basic outline of the approach is the following:

**ContractGraph**\( (G = (V, E)) = \)

1. Identify a set of disjoint connected components in \( G \)
2. Pick a vertex to represent its components and contract other vertices in the same component into it.
3. Let \( V' \) be the set of vertices after contracting each component into its representing vertex.
4. Let \( E' \) be edges after relabeling each edge so its endpoints refer to the representing vertex.
5. Remove self-loops and parallel edges in \( E' \).
6. If \( |E'| > 0 \) then **ContractGraph**\( (G' = (V', E')) \)

We refer to each recursive call as a contraction step.

Now let’s go through some examples of how we might contract a graph. Consider the following graph:
In this graph, we could identify the disjoint components \( \{a, b, c\}, \{d\}, \{e, f\} \).

After contracting, we would be left with a triangle. Note that in the intermediate step, when we join \( a, b, c \), we create redundant edges to \( d \) (each one of them had an original edge to \( d \)). We therefore replace these with a single edge. However, in some algorithms, it is convenient to allow parallel (redundant) edges rather than going to the work of removing them. This is sometimes referred to as a multigraph.

Instead of contracting \( \{a, b, c\}, \{d\}, \{e, f\} \), we could contract the components \( \{a, c\}, \{b, d\}, \{e, f\} \).

In this case, we would be left with three vertices connected in a line. In the two limits, we could contract nothing, or contract all vertices.

There are a couple special kinds of contraction that are worth mentioning:

**Edge Contraction:** Select a disjoint set of edges, and treat each edge as its own component, so only pairs of vertices connected by an edge are contracted (such as contracting the components \( \{a, c\}, \{b, d\}, \{e, f\} \) in the graph above). One can think of the edges as pulling the two vertices together into one and then disappearing.

**Star Contraction:** One vertex of each component is identified as the center of the star and all other vertices are directly connected to it (such as contracting the components \( \{a, b, c\}, \{d\}, \{e, f\} \) in the graph above).

Why, you might ask, is contraction useful in parallel algorithms? Well, it is first worth noting that if the size of the graph reduces by a constant factor on each step, then the algorithm will finish after only \( O(\log n) \) steps. (This is the familiar recurrence \( f(n) = f(n/b) + c \).) Therefore, if we can run each step in parallel, we have a good parallel algorithm. However, even if we can contract in parallel, how can we use it to do anything useful? There are certain properties of the graph that can be maintained through contractions. For example, contraction maintains the connectivity structure of the graph. Therefore, if we start out with \( k \) connected components in a graph, we will end up with \( k \) components. So we can simply keep on contracting the graph until we run out of edges, then we check how many isolated vertices (i.e., vertices with not edges connecting to it) are left in the graph, and that tells us how many connected component there are in the original graph.

So, how do we select components? We must do so in parallel. Also ideally, the size of the vertex set after a contraction step should reduce by a constant fraction. Being able to do this efficiently and deterministically turns out to be quite a difficult problem.
As an example, let’s consider a graph that is simply a cycle of vertices each pair connected by an (undirected) edge. The example below shows a 6-cycle ($C_6$).

We need to decide for each vertex in parallel, whether it would represent a component or whether it would contract into one of its neighbor. How do we make such a decision for each vertex in parallel without too much coordination? It’s difficult to make a decision based on the structure of the graph, because from the point of view of every vertex, the world looks the same. For instance, if every vertex decides to contract into its neighbor to the right, we are never going to make any progress (we simply relabel every vertex, and the new vertex set will have the same size). Put it differently, if a vertex decides to contract into say, its neighbor to the right, that neighbor better not itself contract into some other vertex. We essentially need a way to break symmetry. It turns out that randomization is a huge help. Any ideas how we might use randomization?

Say we want to do edge contraction on the cycle. That is, we want in parallel to select some number of disjoint edges (i.e., that don’t share an endpoint). We could flip a coin for each edge. Then, the rule is: if an edge gets a head and both its neighbors get tails, then select that edge. This guarantees that no two adjacent edges will be selected so our components are disjoint. Now we can contract each edge and we are left with a smaller cycle.

To do this, the easiest thing to do is to represent the graph as an edge list $E$ sorted in the same order edges in the cycle. Thus, in each contraction step, we do the following:

**ContractCycles**($E$) =

1. In parallel, for all edges $e \in E$, flip a coin.
2. Mark an edge as selected if it has a head and its two neighbors have tails.
3. Construct a new edge $E'$ by relabeling the edges $e \in E$. If an edge is selected, we contract the vertex with larger id into the vertex with smaller id. We can relabel all edges in parallel by simply looking at an edge and its neighbors, and relabel vertices that got contracted.
4. Remove self loops in $E'$ (notice that because this is a cycle, there can only be self loops but no parallel edges after relabeling).
5. If ($|E'| > 0$) then repeat **ContractCycles**($E'$).

Each contract step with a cycle of size $n$ can be done with work $\Theta(n)$ and span $\Theta(\lg n)$. For example, we can store the graph as an edge list in an array, sorted in the same order as in the cycle with one duplicate edge per cycle (i.e., same edge at the beginning and end of the array). But what about the overall work and span?

Before we go ahead analyze the work and span, let’s first review some probability theory, because they will come in handy, not just for this lecture but also for materials that we will cover in the rest of the term.
Review of Probability Theory

Recall from probability theory that a sample space is the set \( \Omega \) of all possible outcomes. In the case of flipping coins on \( n \) edges, the size of the sample space is \( 2^n \) representing all possible outcomes, which are all equally likely if the coins are unbiased. An event \( \mathcal{E} \subseteq \Omega \) is any subset of the sample space. In our example, it might consist of all outcomes such that a particular edge is selected (it comes up heads and its neighbors tails), or all outcomes where the number of edges selected is at least \( n/8 \). Assuming each outcome in the sample space is equally likely to happen, we can then ask what the probability of that event is by simply counting the number of outcomes in the event and dividing by the total number of possible outcomes, i.e., \( \Pr\{\mathcal{E}\} = |\mathcal{E}|/|\Omega| \). On the other hand, if they are not equally likely, then you have to re-weigh by the probability of having a particular outcome.

A random variable is a function \( f : \Omega \rightarrow \mathbb{R} \). We typically denote random variables by capital letters \( X, Y, \ldots \). That is, a random variable assigns a numerical value to an outcome \( \omega \in \Omega \). In our algorithm, one useful random variable is the count of number of selected edges. It maps each outcome to a number between 0 and \( n/2 \) (why not \( n \)?). A special kind of random variable, called indicator random variable, maps each outcome to either 0 or 1. For an event \( \mathcal{E} \), the indicator random variable \( \mathbb{I}\{\mathcal{E}\} \) takes on the value 1 if \( \mathcal{E} \) occurs and 0 otherwise. In our example, one can define an indicator random variable to be whether a particular edge is selected; the indicator random variable has value 1 if the edge is selected and has value 0 if the edge is not selected.

The expectation (or expected value) of a random variable is simply the weighted average of the value of the function over all outcomes. The weight is the probability of each outcome, giving

\[
E[X] = \sum_{\omega \in \Omega} X(\omega) \Pr\{\omega\}
\]

in the discrete case (i.e., your sample space contains countable outcomes). Applying this definition to an indicator random variable, we have \( E[\mathbb{I}\{\mathcal{E}\}] = \Pr\{\mathcal{E}\} \).

In our example, we are interested in the expectation that each single edge is selected and the expectation on the total number of edges selected. The expectation that a single edge is selected is \( 1/8 \) since there are 8 possible outcomes on the coin tosses of three neighbors, and only one of them leads to the edge being selected.

Recall that one of the most important rules of probability is linearity of expectations. It says that given two random variables \( X \) and \( Y \), \( E[X] + E[Y] = E[X+Y] \). This does not require that the events are independent. In particular, if the events are selection of edges, two neighboring edges are certainly not independent (why?), but yet we can still add their expectations. So the expectation that two neighboring edges are selected is \( 1/8 + 1/8 = 1/4 \). And hence the expected total number of edges selected is \( n \times 1/8 = n/8 \).

Exercise 1  Come up with a randomized scheme that on a graph that consists of a single undirected cycle of length \( n \) selects an disjoint set of edges of expected size \( 5n/27 \). Can you do even better?
Analyzing Expected Work and Span

Let’s first consider the work. As we computed earlier, assume our coin tosses are non-biased and independent, the expectation that an edge is selected is $1/8$. Let $X_n$ be a random variable indicating how many edges are removed in a given round starting with $n$ edges, $E[X_n] = n/8$.

Assuming we always remove exactly $1/8^{th}$, we would have the following recurrence for the number of rounds

$$R(n) = R(7n/8) + 1.$$ 

Since the size is decreasing geometrically this solves to $O(\log n)$ rounds. Furthermore if we use edge list to represent the graph, each round can be done in $O(\log n)$ span (needed for filtering out the remaining edges), and $O(n)$ work. This would give the work recurrence as

$$W(n) = W(7n/8) + O(n) = O(n).$$

Unfortunately we don’t remove exactly $1/8^{th}$ of the vertices, but the number removed in a given round depends on the toss of the coins.

Let’s call $Y_n$ be a random variable that indicates how many vertices are left in a graph starting out with size $n$ after a round of contraction. We know that $E[Y_n] = 7n/8$, because an edge doesn’t get picked to be contracted with probability of $7/8$.

Recall Markov Inequality, which states that:

**Theorem 1 (Markov Inequality):** Given a random variable $X$ with expectation $E[X] = \mu$, for any $\lambda > 0$,

$$\Pr\{X > (1 + \lambda)\mu\} < \frac{1}{1 + \lambda}$$

Intuitively, one can think of expectation as a weighted average. In Markov Inequality, the $\mu$ specifies a distance from this “average,” and the probability that $X$ has a value much larger than its expectation is low.

In order to show that the work is linear, ideally we’d like to show that a constant fraction of edges are reduced at every round. We know that’s the case in expectation, but that’s just the average. We can use Markov to show that, however, the bad case (i.e., a round that don’t reduce the edges by a constant fraction) occurs infrequently. Thus for the most part, we decrease the number of edges by a constant fraction, and thereby bounding the overall work.

Say we choose $\lambda$ to be $1/14$, then Markov gives us

$$\Pr\left\{Y_n > \left(1 + \frac{1}{14}\right)\frac{7n}{8}\right\} = \Pr\left\{Y_n > \frac{15n}{16}\right\} \leq \frac{14}{15}$$
That means, we can calculate the expected work this way:

\[
W(n) = \sum_i \Pr \{ Y_n = i \} \cdot W(i) + cn
\]

\[
\leq \Pr \left\{ Y_n > \frac{15n}{16} \right\} \cdot W(n) + \Pr \left\{ Y_n \leq \frac{15n}{16} \right\} \cdot W \left( \frac{15n}{16} \right) + cn
\]

\[
\leq \frac{14}{15} W(n) + \frac{1}{15} W \left( \frac{15n}{16} \right) + cn
\]

\[
\frac{1}{15} W(n) \leq \frac{1}{15} W \left( \frac{15n}{16} \right) + cn
\]

\[
W(n) \leq W \left( \frac{15n}{16} \right) + 15cn
\]

\[
= \Theta(n)
\]

We can similarly calculate the span:

\[
\overline{S}(n) = \sum_i \Pr \{ Y_n = i \} \cdot \overline{S}(i) + c\lg n
\]

\[
\leq \Pr \left\{ Y_n > \frac{15n}{16} \right\} \cdot \overline{S}(n) + \Pr \left\{ Y_n \leq \frac{15n}{16} \right\} \cdot \overline{S} \left( \frac{15n}{16} \right) + c\lg n
\]

\[
\leq \cdots
\]

\[
\overline{S}(n) \leq \overline{S} \left( \frac{15n}{16} \right) + 15c\lg n
\]

\[
= \Theta(\lg^2 n)
\]

Note: In lecture, we first analyzed the work using number of rounds based on a lemma from Karp et. al. This analysis is better for the edge contractions with cycles, because we can actually show that the work is linear. The lemma is useful for general graph contraction, and the notes for it have been moved to the next lecture with notes on general graph contraction.