Recitation 10 – *Bellman Ford and Min-Cut with Graph Contraction*

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

October 21st, 2014

1 Announcements

- How's *AbridgedLab*? It's due on Friday!
- *SegmentLab* should be released on Friday.

2 A Brief Review of Bellman-Ford

Recall that in Dijkstra's Algorithm, we maintain a priority queue containing the shortest *known* distance, $d(v)$, to all visited vertices, v, at any given point in time during the algorithm. Additionally, we keep a table to contain the *final* minimum distances, as we discover them.

Q: Why can't we use Dijkstra's algorithm to compute shortest path when there are negative edges?

A:

Q: How can we find shortest paths on a graph with negative edge weights?

A:

2.1 Bellman-Ford Algorithm

Let us define $\delta_G^l(s,t) =$ the shortest weighted path in G from s to t using at most l edges.

Q: What is $\delta_G^0(s,v)$ for all v?

A:

Q: Suppose we know $\delta_G^k(s,v)$ for all v. How can we compute $\delta_G^{k+1}(s,v)$ for all v?

A:

The Bellman-Ford algorithm is based on this idea. We will incrementally compute $\delta_G^0(s,v), \delta_G^1(s,v), \ldots$ for all v.

Q: What is our stopping condition? That is, at what value of l do we stop computing $\delta_G^l(s,v)$?

A:

Q: What happens if we have a negative weight cycle?

A:

Q: Finally, what is the work/span of this algorithm?

A:

3 Randomized Min-Cut With Graph Contraction

3.1 Foreword

This section is a particularly (in my opinion) interesting amalgamation of several topics you have discussed so far in this course, and relates in interesting ways to topics you will soon see. It is a good exercise to go through to gauge your understanding of course material.

However, as the topic of this recitation is graph contraction, it is *most* important that you understand how parallel graph contraction is used in this algorithm, as well as how parallel graph contraction works in general. If you have specific questions about other pieces of the algorithm, I'd be happy to walk you through them outside of recitation.

3.2 Definitions

Suppose G is an undirected, unweighted, connected, simple graph with vertex set V, and edge set E. Note that in all of our analysis, we use that $|E| = m > n = |V|$, as G is connected. In this algorithm, when we contract an edge, we will remove self loops, but not duplicate edges – after contraction, our graph may become a *multigraph*.

- A *cut*, (A,B), of a graph is just a partition of V into two (nonempty) sets A and $B - A \cup B = V$.
- An edge $e = uv$ *crosses* a cut (A,B) if $u \in A$ and $v \in B$ or vice-versa.
- We say that the cut $C = (A,B)$ has *value* $|C| = v$ iff the sum of the weights of those edges which cross is v.
- The *min-cut* of G is a cut of minimum value. Note that the min-cut need not be unique!

Q: Suppose the min-cut $C = (A, B)$ of G has value $|C| = c$. What is a lower bound on the number of edges in G?

A:

3.3 A Really (Really) Crappy Min-Cut Algorithm

Consider the following randomized algorithm that finds the min-cut with some $p > 0$ probability (yes, this p is *very* small, but we will address this problem later):

```
1  fun CONTRACT(G = (V, E)) =
2    if (|V| = 2) then G else
3    let
4      val e = an edge from E picked uniformly at random
5      val G' = contract e in G, removing self-loops, but keeping multiedges
6    in CONTRACT(G')
```

Q: This algorithm outputs a multigraph on two vertices. How do we turn this into our (possibly minimum) cut? What is the size of this cut?

A: Suppose the output produces a multigraph, G', on two vertices, s and t. Let S be the set of vertices in G which were contracted into s, and T be the same for t. Then our cut should be (S, T), and the size of this cut is the number of (multi-)edges in G'. This should be straightforward to verify (Hint: first show that after one contraction, any cut of the contracted graph represents a cut in the original).

Q: How fast does this algorithm run?

A: This boils down to how fast we can contract an edge. If we use an STSequence representation, contracting an edge takes $O(m)$ work and $O(1)$ span – see the lecture notes for more information. Since every contraction reduces the number of vertices by 1, there are at most n iterations, yielding a total of $O(mn)$ work and $O(n)$ span. This is far too slow!

Q: How could we possibly make our algorithm faster?

A: If we could somehow contract a bunch of edges in the same step, we could just use our parallel graph contraction techniques from lecture!

3.4 Making It Fast, Making It Parallel

Suppose now, in an attempt to be able to contract many edges at once, we modify the algorithm slightly. Instead of choosing a random edge to contract *at each iteration*, we will begin the algorithm by randomly permuting all the edges in G. We then contract the edges in the order of this permutation.

By the end of the algorithm, we should have contracted some specific *prefix* of this permutation, leaving us with exactly two vertices – contracting a larger prefix will leave us with only one vertex, and contracting a smaller prefix will leave us with more than two.

Q: Suppose we know what prefix, $P \subseteq E$, of our permutation to contract to leave two vertices. How fast can we do the contraction?

A: Let $G[P]$ be the graph G with only the edges from P. Then we can fully contract $G[P]$ using tree contraction in $O(|P|) \in O(m) = O(m)$ work and $O(\log^2 n)$ span. That's great!

Q: What simple algorithms may help us determine which prefix, P, of our permutation to contract in order to be left with 2 vertices?

Hint: contracting a larger prefix yields fewer vertices, and contracting a smaller prefix yields more, so we can think of our prefixes as sorted *by the number of vertices left after contraction. We would like to find the length of the prefix which leaves exactly 2 vertices after contraction.*

A: We must do a binary search over the prefixes, where each comparison involves looking at the number of vertices left after contraction and comparing that value to 2.

Q: How fast is this prefix-finding algorithm?

A: There are m prefixes of the permutation of edges, so the binary search takes $O(\log m) \in O(\log n)$ steps. Each step involves contracting a set of edges. We just determined that fully contracting some subset of the edges can be done in $O(m)$ work and $O(\log^2 n)$ span with tree contraction, so the total work is $O(m \log n)$ and span is $O(\log^3 n)$.

So we have the total work of our faster, parallel, crappy, randomized min-cut algorithm to be $O(m \log n)$ and the span is $O(\log^3 n)$. Better! *Note that this algorithm is not optimal – we can reduce the work to $O(m)$ by using a smarter prefix-finding algorithm.*

3.5 Making It Not Crappy – Fun With Probability! [OPTIONAL]

In the field of randomized algorithms, a *Monte Carlo* algorithm is an algorithm which yields the correct answer with some probability $p > 0$. Even if this p is small, we can independently run the algorithm many times in order to increase our probability of finding the correct answer!

First, let us determine the probability that our algorithm outputs a given minimum cut (the correct answer). Suppose that the minimum cut in G has size c. The key thing to note is that our algorithm outputs a specific minimum cut, C, iff none of the edges which cross C are contracted (why?).

Q: What is a lower bound on the number of edges in a graph G with r vertices and min-cut of size c

A: From way up above, $\frac{rc}{2}$!

Q: Suppose that the minimum cut has size c. Suppose that we have partially run the algorithm and have contracted the graph down to r vertices, but we have not contracted any of the edges crossing the minimum cut. What is the probability that we will contract a min-cut edge in the next step?

A: In order to contract a min-cut edge, we must choose from the $\geq \frac{rc}{2}$ edges, one of the c min-cut edges to contract. This is done with probability *at most* $\frac{c}{rc/2} = \frac{2}{r}$.

Q: So then what is the probability that we never contract a min-cut edge throughout the course of the algorithm?

A: The probability when we have r vertices left, and no min-cut edge has already been contracted, that we don't contract a min-cut edge is $1 - \frac{2}{r}$. Thus the probability that we never contract a min-cut

edge is

$$\left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\cdots\left(1-\frac{2}{3}\right) = \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right)$$
$$= \binom{n}{2}^{-1}$$

Q: Suppose we run this algorithm $n^2 \ln\left(\frac{1}{\delta}\right)$ times. What is the probability that we never find the min-cut?

A: The approximatimation (the inequality below) requires some trickiness with logarithms:

$$\left(1-\binom{n}{2}^{-1}\right)^{n^2\ln(1/\delta)} \leq e^{-\binom{n}{2}^{-1} n^2 \ln(1/\delta)} \approx \delta$$

So we have that if we run the algorithm $n^2 \ln(1/\delta)$ times, the chance that we get the wrong answer is at most δ. If we make δ a small constant (not dependent on n), such as 0.01, then the total work is $O(mn^2 \log n)$, and the span is $O(\log^3 n)$ (we can do all the independent runs in parallel). Then we have found the min-cut with 99% certainty!

3.6 Wrap-Up

We have found an algorithm that finds the min-cut with a given amount of certainty (as long as the desired precision is constant). This algorithm is *fast* (we can improve the work to $O(mn^2)$), and more importantly, the algorithm is *parallel* ($O(\log^2 n)$ span) – this is a big deal, because the traditional deterministic algorithms are not.

There are lots of extra things you can do with this algorithm (improve the space utilization, improve the work of running the algorithm lots of times by sharing work between the runs, reduce parts of this problem to well-known minimum spanning tree algorithms) – resources can be provided on Piazza.