Lecture 13 — Graph Contraction, Connectivity

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — October 11, 2011

Announcements:
- Exam 1 back at end of class
- Assignment 5 due Oct. 18

Today:
- Dijkstra Costs
- Graph Contraction

1 Cost of Dijkstra

Most of the class did not get the short question about the cost of Dijkstra's algorithm when using arrays. We probably did not cover the cost of Dijkstra's algorithm in enough detail, so we will review it here. Dijkstra's algorithm basically operates on three data structures: (1) a structure for the graph itself, (2) a structure to maintain the distance to each vertex that has already been visited, and (3) a priority queue holding distances of vertices that are neighbors of the visited vertices.

Here is Dijkstra's algorithm with the operations on these data types in boxes.

```
1  fun dijkstra(G, s) =
2  let
3    fun dijkstra'(D, Q) =
4      case [PQ.deleteMin](Q) of
5        (PQ.empty, _) ⇒ D
6      | ((d, v), Q') ⇒
7        if ([(v, _) ∈ D]) then dijkstra'(D, Q')
8        else let
9          fun relax (Q, (u, w)) = [PQ.insert](d + w, u) Q
10         val N = [N_G(v)]
11         val Q'' = iterate relax Q' N
12       in dijkstra'([D ∪ {(v, d)}], Q'') end
13 in
14   dijkstra'({}, PQ.insert(∅, (0, s)))
15 end
```

The `PQ.insert` in Line 14 is called only once, so we can ignore it. Of the remaining operations, Lines 10 and 11 are on the graph, Lines 7 and 12 are on the table of visited vertices, and Lines 4 and

9 are on the priority queue. For the priority queue operations, we have only discussed one cost model, which for a queue of size n requires $O(\log n)$ for each of `PQ.insert` and `PQ.deleteMin`. We have no need for a `meld` operation here. For the graph, we can either use a tree-based table or an array to access the neighbors[1] There is no need for single threaded array since we are not updating the graph. For the table of distances to visited vertices we can use a tree table, an array sequence, or a single threaded array sequences. The following table summarizes the costs of the operations, along with the number of calls made to each operation. There is no parallelism in the algorithm so we only need to consider the sequential execution of the calls.

Operation	Line	# of calls	PQ	Tree Table	Array	ST Array
`deleteMin`	4	$O(m)$	$O(\log n)$	-	-	-
`insert`	9	$O(m)$	$O(\log n)$	-	-	-
Priority Q total			$O(m \log n)$	-	-	-
`find`	7	$O(m)$	-	$O(\log n)$	$O(1)$	$O(1)$
`insert`	12	$O(n)$	-	$O(\log n)$	$O(n)$	$O(1)$
Distances total			-	$O(m \log n)$	$O(n^2)$	$O(m)$
$N_G(v)$	10	$O(n)$	-	$O(\log n)$	$O(1)$	-
`iterate`	11	$O(m)$	-	$O(1)$	$O(1)$	-
Graph access total			-	$O(m + n \log n)$	$O(m)$	-

We can calculate the total number of calls to each operation by noting that the body of the let starting on Line 8 is only run once for each vertex. This means that Lines 10 and 12 are only called $O(n)$ times. Everything else is done once for every edge.

Based on the table one should note that when using either tree tables or single threaded arrays the cost is no more than the cost of the priority queue operations. Therefore there is no asymptotic advantage of using one over the other, although there might be a constant factor speedup that is not insignificant. One should also note that using regular purely functional arrays is not a good idea since then the cost is dominated by the insertions and the algorithm runs in $\Theta(n^2)$ work.

2 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 is parallelism in BFS since each level can be explored in parallel, assuming the number of levels is not too large. However, there was no parallelism in DFS. There was also no parallelism in the version of Dijkstra's algorithm we discussed, which used priority first search.[2] 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.

[1] We could also use a hash table, but we have not yet discussed them.

[2] In reality there is some parallelism in both DFS and Dijkstra when graphs are dense—in particular, although vertices need to visited sequentially the edges can be processed in parallel. If we have time, we will get back to this when we cover priority queues in more detail.

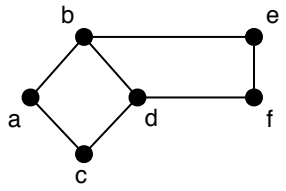
The first of these techniques is graph contraction. This is actually a reasonably simple technique and can be applied to a variety of problems including graph connectivity, spanning trees, and minimum spanning trees. In the discussion of graph contraction, we will assume that the graph is undirected unless otherwise stated. The basic outline of the approach is the following:

ContractGraph$(G = (V, E))$ =

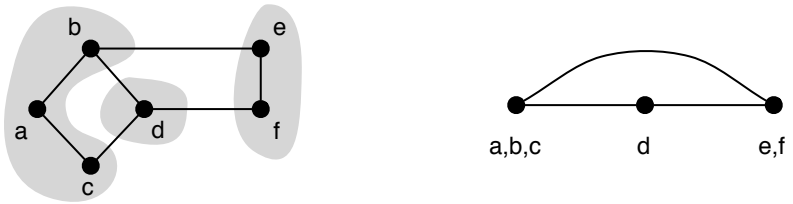
1. Identify a set of disjoint connected components in G
2. V' = the set of vertices after contracting each component into a single vertex
3. E' = after relabeling each edge so its endpoints refer to the new vertex
4. E'' = remove self-loops (and parallel edges)
5. If $(|E''| > 0)$ then **ContractGraph**$(G' = (V', E''))$

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: Only pairs of vertices connected by an edge are contracted. One can think of the edges as pulling the two vertices together into one and then disappearing.

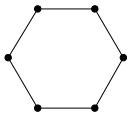
Star Contraction: One vertex of each component is identified as the center of the star and all other vertices are directly connected to it.

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 $f(n) = f(n/2) + c$.) Therefore, if we can run each step in parallel, we have a good parallel algorithm. In fact, such algorithms can be theoretically much more parallel than Breadth First Search since the parallelism no longer depends on the diameter of the graph. However, even if we can contract in parallel, how can we use it to do anything useful? One reason is that 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. It also turns out that if we pick the edges to contract on carefully, then the contraction maintains other properties that are useful. For example, we will see how it can be used to find minimum spanning trees.

After answering how contraction is useful, the next question to ask is: how do we select components? Remember that we would like to do this in parallel. Let's start by limiting ourselves to edge contractions. Any ideas?

In particular in parallel we want to select some number of disjoint edges (i.e that don't share an endpoint). Ideally, the edges should cover a constant fraction of the vertices so that the graph contracts sufficiently.

Being able to do this efficiently and deterministically turns out to be quite a difficult problem. The issue is that from the point of view of every vertex the world might look the same. 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).



Wherever we are on the cycle, it looks the same, so how do we decide how to hook up? As an example, if each node tries to join the person to the right, we are not going to make any progress. We essentially need a way to break symmetry. It turns out that randomization is a huge help. Any ideas how we might use randomization?

On a cycle, we could flip a coin for each edge. Then, the rule is: *if an edge gets a heads and both its neighbors*[3]*, 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.

How much do we expect this graph to contract? Let's assume that all coin flips are unbiased and are independent. What is the expectation that an edge is selected? Notice that the probability that the edge comes up heads and both neighbors come up tails is $\frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} = \frac{1}{8}$. To calculate the expected number of edges that are removed, the easiest way is to apply linearity of expectations. In particular, the expectation that each edge is selected is $1/8$, so if we have a cycle of length n (which has n vertices and n edges), then the expected number of edges that are removed is $n/8$.

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 $n/3$.*

[3] of which there are 2 in a cycle