Lecture 16 — k^{th} Smallest and Graph Contraction

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Margaret Reid-Miller — 7 March 2013

What was covered in this lecture:

- Another randomized algorithm: `kthSmallest`
- Introduction to graph contraction and its application to connectivity.

1 Finding the k^{th} Smallest Element

Now we will generalize the idea of finding the second largest element to finding the the k^{th} smallest element in a sequence. Again, the algorithm will use randomization, and we will use indicator random variables for our analysis. But this time we consider how to analyze a recursive algorithm.

Consider the following problem:

> **Input:** S — a sequence of n numbers (not necessarily sorted)
> **Output:** the k^{th} smallest value in S (i.e. `(nth (sort S) k)`).
> *Requirement:* $O(n)$ expected work and $O(\log^2 n)$ span.

Note that the linear-work requirement rules out the possibility of sorting the sequence. Here's where the power of randomization gives a simple algorithm.

```
1  function kthSmallest(k, S) = let
2        p = a value from S picked uniformly at random
3        L = ⟨ x ∈ S | x < p ⟩
4        R = ⟨ x ∈ S | x > p ⟩
5    in if (k < |L|) then kthSmallest(k, L)
6       else if (k < |S| − |R|) then p
7       else kthSmallest(k − (|S| − |R|), R)
```

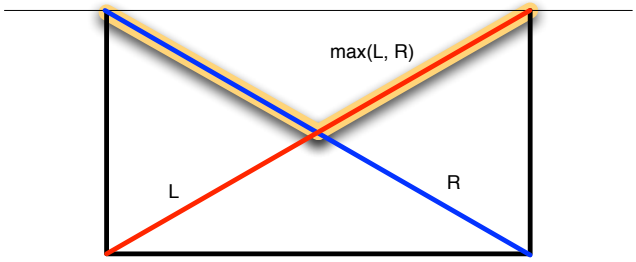
We'll try to analyze the work and span of this algorithm. Let $X_n = \max\{|L|, |R|\}$, which is the size of the larger side. Notice that X_n is an upper bound on the size of the side the algorithm actually recurses into. Now since Step 3 is simply two `filter` calls, we have the following recurrences:

$$\begin{aligned} W(n) &= W(X_n) + O(n) \\ S(n) &= S(X_n) + O(\log n) \end{aligned}$$

Let's first look at the work recurrence. Specifically, we are interested in $\mathbf{E}[W(n)]$. First, let's try to get a sense of what happens in expectation.

How big is $\mathbf{E}[X_n]$*?* To understand this, let's take a look at a pictorial representation:

†Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.



The probability that we land on a point on the curve is $1/n$, so

$$\mathbf{E}\left[X_n\right] = \sum_{i=1}^{n-1} \max\{i, n-i\} \cdot \tfrac{1}{n} \leq \sum_{j=n/2}^{n-1} \frac{2}{n} \cdot j \leq \frac{3n}{4}$$

(Recall that $\sum_{i=a}^{b} i = \frac{1}{2}(a+b)(b-a+1)$.)

Aside: This is a counterexample showing that $\mathbf{E}\left[\max\{X,Y\}\right] \neq \max\{\mathbf{E}\left[X\right], \mathbf{E}\left[Y\right]\}$.

This computation tells us that in expectation, X_n is a constant fraction smaller than n, so we should have a nice geometrically decreasing sum, which works out to $O(n)$. Let's make this idea concrete: Suppose we want each recursive call to work with a constant fraction fewer elements than before, say at most $\frac{3}{4}n$.

What's the probability that $\mathbf{Pr}\left[X_n \leq \frac{3}{4}n\right]$*?* Since $|R| = n - |L|$, $X_n \leq \frac{3}{4}n$ if and only if $n/4 < |L| \leq 3n/4$. There are $3n/4 - n/4$ values of p that satisfy this condition. As we pick p uniformly at random, this probability is

$$\frac{3n/4 - n/4}{n} = \frac{n/2}{n} = \frac{1}{2}.$$

Notice that given an input sequence of size n, how the algorithm performs in the future is irrespective of what it did in the past. Its cost from that point on only depends on the random choice it makes after that. So, we'll let $\overline{W}(n) = \mathbf{E}\left[W(n)\right]$ denote the expected work performed on input of size n.

Now by the definition of expectation, we have

$$\begin{aligned}
\overline{W}(n) &\leq \sum_i \mathbf{Pr}\left[X_n = i\right] \cdot \overline{W}(i) + c \cdot n \\
&\leq \mathbf{Pr}\left[X_n \leq \tfrac{3n}{4}\right] \overline{W}(3n/4) + \mathbf{Pr}\left[X_n > \tfrac{3n}{4}\right] \overline{W}(n) + c \cdot n \\
&= \tfrac{1}{2}\overline{W}(3n/4) + \tfrac{1}{2}\overline{W}(n) + c \cdot n \\
&\Longrightarrow (1 - \tfrac{1}{2})\overline{W}(n) = \tfrac{1}{2}\overline{W}(3n/4) + c \cdot && \text{[collecting similar terms]} \\
&\Longrightarrow \overline{W}(n) \leq \overline{W}(3n/4) + 2c \cdot n. && \text{[multiply by 2]}
\end{aligned}$$

In this derivation, we made use of the fact that with probability $1/2$, the instance size shrinks to at most $3n/4$—and with probability $1/2$, the instance size is still larger than $3n/4$, so we pessimistically upper bound it with n. Note that the real size might be smaller, but we err on the safe side since we don't have a handle on that.

Finally, the recurrence $\overline{W}(n) \leq \overline{W}(3n/4) + 2cn$ is root dominated and therefore solves to $O(n)$.

1.1 Span

Let's now turn to the span analysis. We'll apply the same strategy as what we did for the work recurrence. We have already established the span recurrence:

$$S(n) = S(X_n) + O(\log n)$$

where X_n is the size of the larger side, which is an upper bound on the size of the side the algorithm actually recurses into. Let $\overline{S}(n)$ denote $\mathbf{E}[S(n)]$. As we observed before, how the algorithm performs in the future is irrespective of what it did in the past. Its cost from that point on only depends on the random choice it makes after that. So then, by the definition of expectation, we have

$$\begin{aligned}
\overline{S}(n) =& \leq \sum_i \mathbf{Pr}\left[X_n = i\right] \cdot \overline{W}(i) + c\log n \\
&\leq \mathbf{Pr}\left[X_n \leq \tfrac{3n}{4}\right] \overline{S}(3n/4) + \mathbf{Pr}\left[X_n > \tfrac{3n}{4}\right] \overline{S}(n) + c \cdot \log n \\
&\leq \tfrac{1}{2}\overline{S}(3n/4) + \tfrac{1}{2}\overline{S}(n) + c \cdot \log n \\
&\implies (1 - \tfrac{1}{2})\overline{S}(n) \leq \tfrac{1}{2}\overline{S}(3n/4) + c\log n \\
&\implies \overline{S}(n) \leq \overline{S}(3n/4) + 2c\log n,
\end{aligned}$$

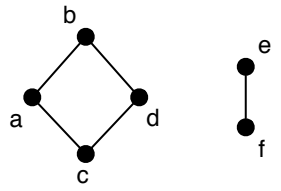
which we know is balanced and solves to $O(\log^2 n)$.

2 Graph Contraction

Until recently, we have been talking mostly about 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 there is parallelism in BFS because each level can be explored in parallel, assuming the number of levels is not too large. But 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.[1] There was plenty of parallelism in the Bellman-Ford algorithm. We are now going to discuss a technique called "graph contraction" that you can add to your toolbox for parallel algorithms. The technique will use randomization.

Graph Connectivity. To motivate graph contraction consider the graph connectivity problem. Two vertices are connected in an undirected graph if there is a path between them. A graph is connected if every pair of vertices is connected. The *graph connectivity* problem is to partition an undirected graph into its maximal connected subgraphs. By partition we mean that every vertex has to belong to one of the subgraphs and the vertices within a partition are connected, and by maximal we mean that no connected vertex can be added to any partition—i.e. there are no edges between the partitions. For example for the following graph:

[1] 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.



graph connectivity will return the two subgraphs consisting of the vertices $\{a, b, c, d\}$ and $\{e, f\}$.

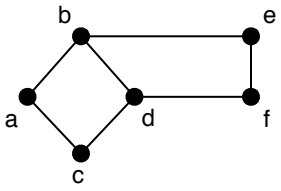
How might we solve the graph connectivity problem? We could do it with a bunch of graph searches. In particular we can start at any vertex and search all vertices reachable from it to create the first component, then move onto the next vertex and if it has not already been searched search from it to create the second component. We then repeat until all vertices have been checked. Either BFS or DFS can be used for the individual searches. This is a perfectly sensible sequential algorithm. However, it has two shortcomings from a parallelism perspective even if we use a parallel BFS. Firstly each parallel BFS takes span proportional to the diameter of the component (the longest distance between two vertices) and in general this could be as large as n. Secondly even if each component is has a small diameter, we have to iterate over the components one by one.

We are therefore interested in an approach that can identify the components in parallel. Furthermore the span should be independent of the diameter, and ideally polylogarithmic in $|V|$. 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 borrow some ideas from our algorithm for the `scan` operation. In particular we will resort to *contraction*—on each round of contraction we will shrink the size of the graph by a constant fraction and then solve the connectivity problem on the contracted graph. Contracting a graph, however, is a bit more complicated than just pairing up the odd and even positioned values as we did in the algorithm for `scan`.

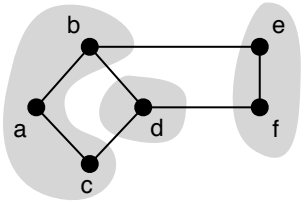
This approach is called *graph contraction*. It is a reasonably simple technique and can be applied to a variety of problems, beyond just connectivity, including spanning trees and minimum spanning trees. We assume the graph is undirected. Lets start by considering a function:

$$\texttt{contract} : \texttt{graph} \rightarrow \texttt{partition}$$

which takes a undirected graph $G = (V, E)$ and returns a partitioning of the vertices of V into connected subgraphs, but not necessarily maximally connected subgraphs (i.e. there can still be edges between the partitions). For example `contract` on the following graph:

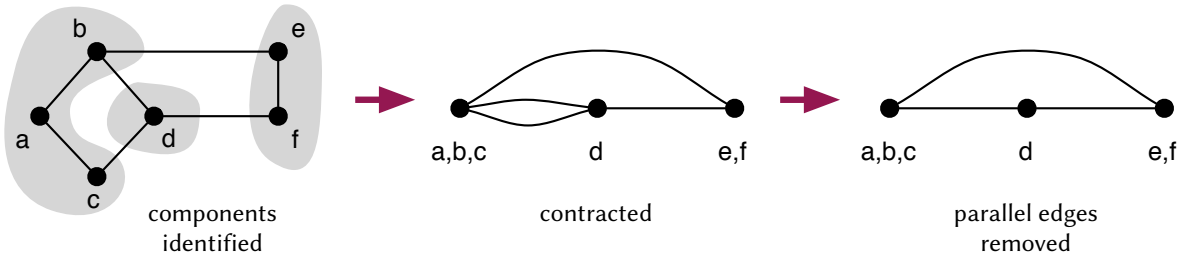


might return the partitioning $\{\{a, b, c\}, \{d\}, \{e, f\}\}$ as indicated by the following shaded regions:



Note that each of the three partitions is connected by edges within the partition. The `contract` function 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.

We will return to how to implement `contract`. But given that the partitions returned by `contract` are not maximal (i.e. there are still edges between them), how might we use `contract` to solve the graph connectivity problem? As we already suggested, we can apply it recursively. To do this on each round we replace each partition with a single vertex and then relabel all the edges with the new vertex name. This can be illustrated as follows:

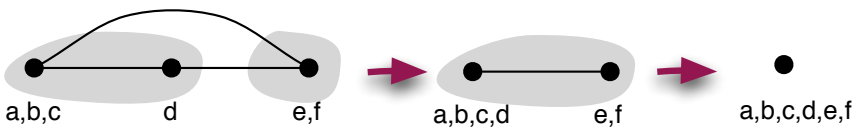


After contracting, we are left with a triangle. Note that in the intermediate step, when we join a,b,c, we create redundant edges to d (both b and c had an original edge to d). We therefore replace these with a single edge. However, depending on the particular use, in some algorithms, it is convenient to allow parallel (redundant) edges rather than going to the work of removing them. Also note that some edge are within a partition. These become self loops, which we drop.

It should be clear that if the graph actually contracts on each round (not all vertices are singleton partitions) then eventually every maximal connected component in the graph will shrink down to a single vertex. This leads to the following high-level algorithm for connectivity:

```
while there are edges left:
  contract the graph
  update the edges to use remaining vertices and remove self loops
```

Continuing with our example, after the first contraction shown earlier we might make the following two contractions in the next two rounds, at which point we have no more edges:



Note that if the input is not connected we would be left with multiple vertices when the algorithm finishes, one per component.

To make this algorithm more concrete we need some representation for the partitions. One way to do this is to use one of the original vertices of each partition as a representative and then keep a mapping from each vertex to this representative. Therefore the partition $\{\{a,b,c\},\{d\},\{e,f\}\}$ could be represented as $(\{a,d,e\},\{a \mapsto a, b \mapsto a, c \mapsto a, d \mapsto d, e \mapsto e, f \mapsto e\})$. With this representation first consider the following more concrete algorithm that counts the number of maximal connected components.

```
1  function numComponents((V, E), i) =
2  if |E| = 0 then |V|
3  else let
4      (V', P) = contract((V, E), i)
5      E' = {(P[u], P[v]) : (u, v) ∈ E | P[u] ≠ P[v]}
6  in
7    numComponents((V', E'), i + 1)
8  end
```

The i is just the round number—as we will see it will be useful in `contract`. As in the high-level description, `numComponents` contracts the graph on each round. Each contraction on Line 4 returns the contracted vertices V' and a table P mapping every $v \in V$ to a $v' \in V'$. Line 5 then does two things. Firstly it 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 and we just count them with $|V|$.

To make the code return the partitioning of the connected components is not much harder, as can be seen by the following code:

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

This code returns the partitioning in the same format as the partitioning returned by `contract`. The only difference of this code from `numComponents` is that on the way back up the recursion we update the representatives for V based on the representatives from V' returned by the recursive call. Consider our example graph. As before, let's say the first `contract` 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 `components` will map all vertices in V' to the same vertex. Suppose this vertex is "a" giving:

$$P' \quad = \quad \{a \mapsto a, d \mapsto a, e \mapsto a\}$$

Now what Line 7 in the code does is for each vertex $v \in V$, it looks for v in P to find its representative $v' \in V'$, and then looks for v' in P' to find its representative in the connected component. This is implemented as $P'[P[v]]$. For example vertex f finds e from P and then looks this up in P' to find a. The final result returned by `components` is:

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

The base case of the `components` algorithm is to label every vertex with itself.