# Recitation 10 — Graph Contraction

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

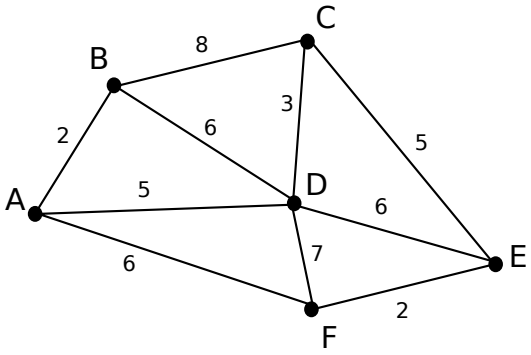
*October 31, 2012*

**Today's Agenda:**

- Prim's algorithm
- Parallel MST
- Lower bounds for merging

## 1 Prim's Algorithm

Prim's algorithm is an algorithm for determining the minimal spanning tree in a connected graph.

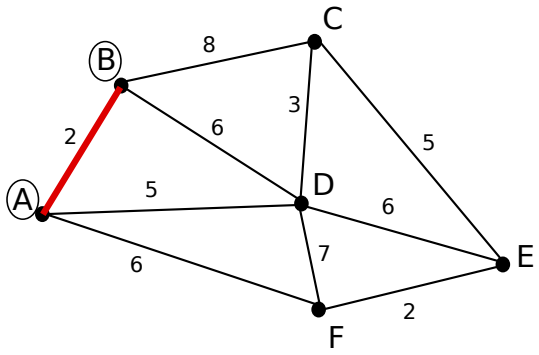



Algorithm:

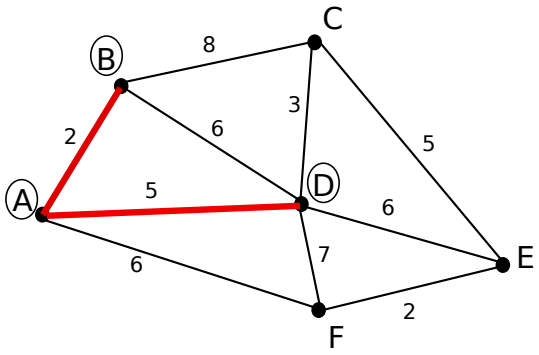
- Choose any starting vertex. Look at all edges connecting to the vertex and choose the one with the lowest weight and add this to the tree.
- Look at all edges connected to the tree that do not have both vertices in the tree. Choose the one with the lowest weight and add it to the tree.
- Repeat step 2 until all vertices are in the tree.

Steps:

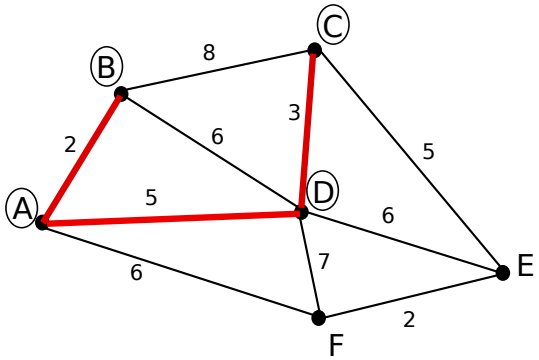
- Choose vertex A. Choose edge with lowest weight: (A,B).



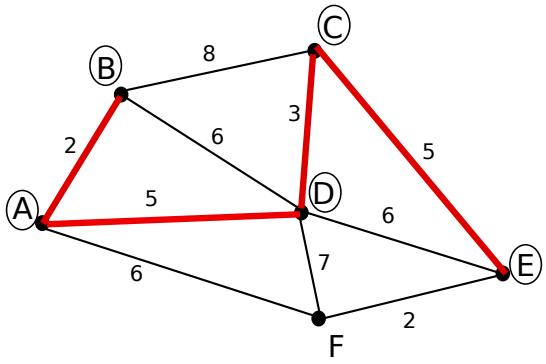

- Look at all edges connected to A and B: (B,C), (B,D), (A,D), (A,F). Choose the one with minimum weight and add it to the tree: (A,D).



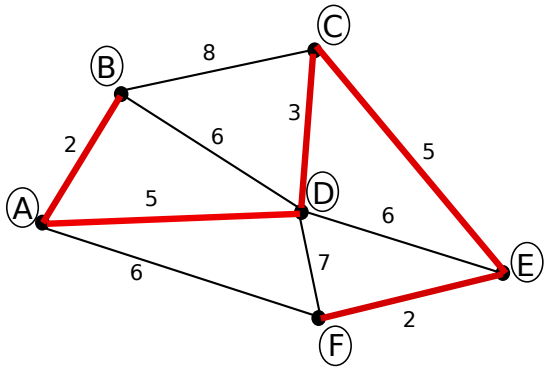

- Look at all edges connected to the tree: (B,C), (D,C), (D,E), (D,F), (A,F). We don't need to consider edge (B,D) because both B and D are in the tree already. We choose edge (D,C).




- Look at all edges connected to A, B, C and D. We still have to connect E and F to the tree. So we look at the edges connected to those and choose the one with the lowest weight: (C,E).




- There is only one vertex F to add before we have a connected minimum spanning tree. We choose edge (E,F) and add that one to the tree.




The tree is now connected and spans all vertices in the graph.

Q: What is the theorem about MSTs and cut edges that applies to Prim's algorithm?

A: The theorem of the light edge rule (give below):

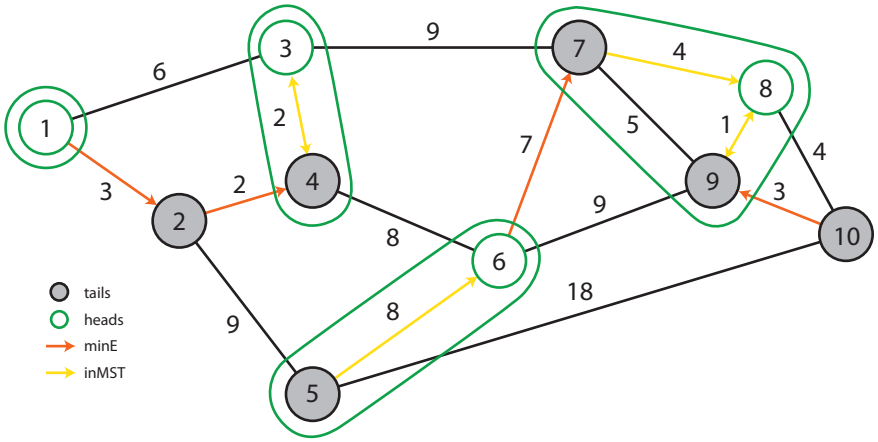
The following theorem states that the lightest edge across a cut is in the MST of $G$:

**Theorem 1.1.** *Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights. For any nonempty $U \subsetneq V$, the minimum weight edge $e$ between $U$ and $V \setminus U$ is in the minimum spanning tree MST(G) of $G$.*

## 2 MST

In lecture 18 we presented a way to write parallel code to find an MST using the algorithm. The idea is simple; instead of contracting any of the edges, only contract edges which are minimum weight from each vertex. Why does this work? Recall the Light Edge Rule / Cut Property from lecture.

Let's go over an example:



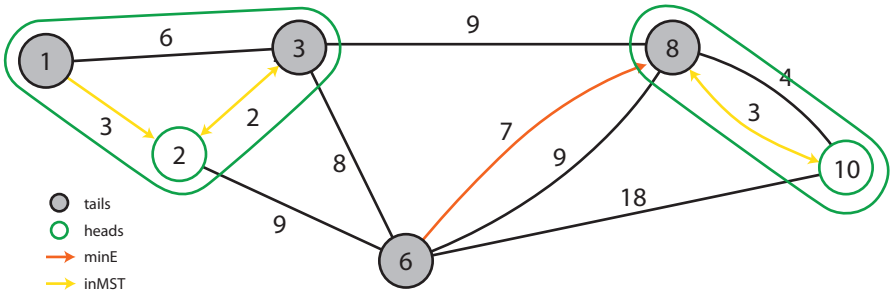

In the first round of the algorithm, we have the following flips:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| H | T | H | T | T | H | T | H | T | T |

Notice that vertices 3 and 4 are contracted, but 1 and 2 are not. Why?

A key point to note here is that in our version of the algorithm, we only consider the minimum *out*-edges from every vertex. That is to say, even though the input graph is undirected, we only pick an edge to be in our MST if it goes from tails to heads. This works because we represent undirectedness by having an edge in both directions.

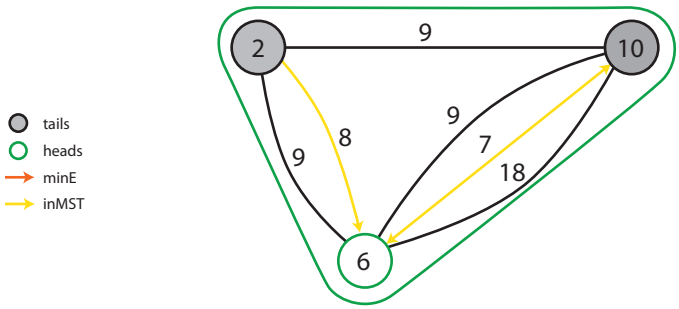
We get the following contracted graph in the next round:




Here is the sequence of flips generated for the second round:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| T | H | T | | | T | | T | | H |

We will generate another sequence of 10 flips here, but we only look at the ones generated for the vertices which remain in our contracted graph. This gives us:




with the following sequence of flips (ignoring vertices not in our graph):

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| | T | | | | H | | | | T |

Of course, the flips seem a little fortuitous, allowing us to contract this graph in 3 rounds. That's because they were made up for this example. In general, it's not unreasonable to have a round of flips which results in no contractions at all. This is where expectation comes in.

Recall from lecture that each vertex has a minimum edge out which contracts with probability $1/4$. By Linearity of Expectation, $n/4$ vertices in expectation will be removed in each round.

## 3 A Merging Lower Bound

This is section 4.5 from Lecture 19.

Closely related to the sorting problem is the merging problem: Given two sorted sequences $A$ and $B$, the merging problem is to combine these sequences into a sorted one. To apply the argument we used for sorting, we'll need to count how many possible outcomes the comparison operation can produce.

Suppose $n = |A|$, $m = |B|$, and $m \leq n$. We'll also assume that these sequences are made up of unique elements. Now observe that we have not made any comparison between elements of $A$ and $B$. This means any interleaving sequence $A$'s and $B$'s elements is possible. Therefore, the number of possible merged outcomes is the number of ways to choose $m$ positions out from $n+m$ positions to put $B$'s elements; this is simply $\binom{n+m}{m}$. Hence, we'll need, in the worst case, at least $\log_2 \binom{n+m}{m}$ comparison queries to merge these sequences.

The following lemma gives a simple lower bound for $\binom{n}{r}$, so that we can simplify $\log_2 \binom{n+m}{m}$ to an expression that we recognize.

**Lemma 3.1** (Binomial Lower Bound)**.**

$$\binom{n}{r} \geq \left(\frac{n}{r}\right)^r.$$

*Proof.* First, we recall that

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{n(n-1)(n-2)\ldots(n-r+1)}{r(r-1)(r-2)\ldots 1} = \prod_{i=0}^{r-1} \frac{n-i}{r-i}.$$

We'll argue that for $0 \leq i < \min(r,n)$, $\frac{n-i}{r-i} \geq \frac{n}{r}$. Notice that

$$\frac{n-i}{r-i} \geq \frac{n}{r} \iff r(n-i) \geq n(r-i) \iff rn - ri \geq nr - ni \iff (n-r)i \geq 0.$$

Therefore, we have $\binom{n}{r} \geq \prod_{i=0}^{r-1} \frac{n-i}{r-i} \geq (n/r)^r$. □

With this lemma, we conclude that the number of comparison queries needed to merge sequences of lengths $m$ and $n$ ($m \leq n$) is at least

$$\log_2 \binom{n+m}{m} \geq m \log_2(1 + \frac{n}{m}),$$

proving Theorem 3.2 (from Lecture 19) given below.

**Theorem 3.2.** *Merging two sorted sequences of lengths $m$ and $n$ ($m \leq n$) requires at least*

$$m \log_2(1 + \tfrac{n}{m})$$

*comparison queries in the worst case.*

Note: With regard to the worst case lower bounds, we are not referring to a **particular** merge sort algorithm, we are only referring to some way of sorting by merging. In general, when calculating lower bounds, we do not stick to a particular algorithm.

So for example, if a merge algorithm gets as input the sequences (1,2,3) and (5,6,7), the algorithm could just look at the last element of the first sequence and the first element of the second sequence, see that it can just append the two sequences and get the final sorted sequence. This algorithm will only make one comparison. So this algorithm, for this input, **does not** give us the worst lower bound. But there might be other algorithms that give us a worst lower bound for this input.

So by calculating the above lower bound, we showed that **no matter** what algorithm we choose for the merging, in the worst case it won't make more than that number of comparisons.

On the other hand, when we want to show upper bounds though we can do a constructive proof. We can find an algorithm for merging and say that the upper bounds of merging in general are not bigger than the upper bound of that algorithm (since we have an algorithm with exactly that upper bound to show for it).