Recitation 10 — More Graph Contraction

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

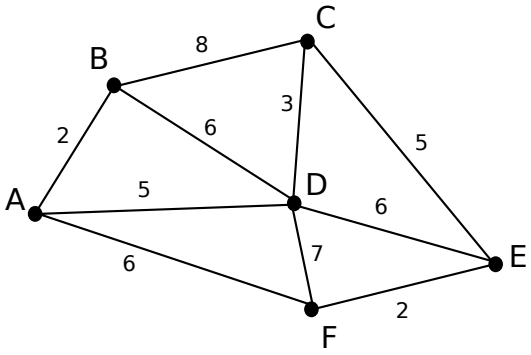
March 27, 2013

Today's Agenda:

- Prim's algorithm
- Boruvka's algorithm
- List Ranking

1 Prim's Algorithm

Prim's algorithm is an algorithm for determining a minimum spanning tree (MST) in a connected graph.

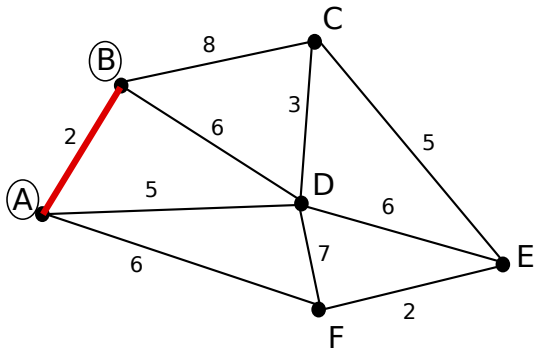


Algorithm:

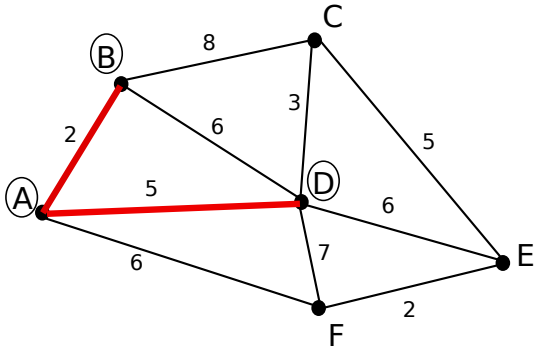
1. Choose any starting vertex. Look at all edges connecting to the vertex and choose one of the ones with the lowest weight and add this to the tree.
2. Look at all edges connected to the tree that do not have both vertices in the tree. Choose the one with the lowest weight and add it to the tree.
3.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

Steps:

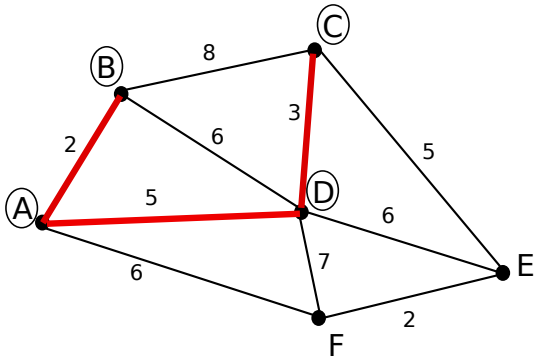
- Choose vertex A. Choose edge with lowest weight: (A,B).



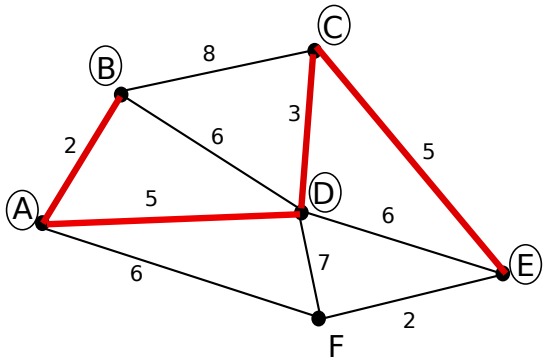
- Look at all edges connected to A and B: (B,C), (B,D), (A,D), (A,F). Choose the one with minimum weight and add it to the tree: (A,D).



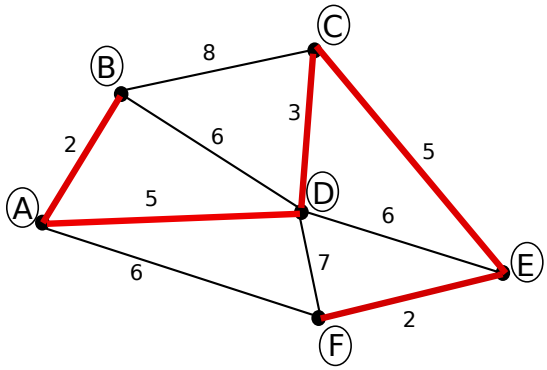
- Look at all edges connected to the tree: (B,C), (D,C), (D,E), (D,F), (A,F). We don't need to consider edge (B,D) because both B and D are in the tree already. We choose edge (D,C).



- 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 rule about MSTs and cut edges that applies to Prim's algorithm?

A: The light edge rule:

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 of G.*

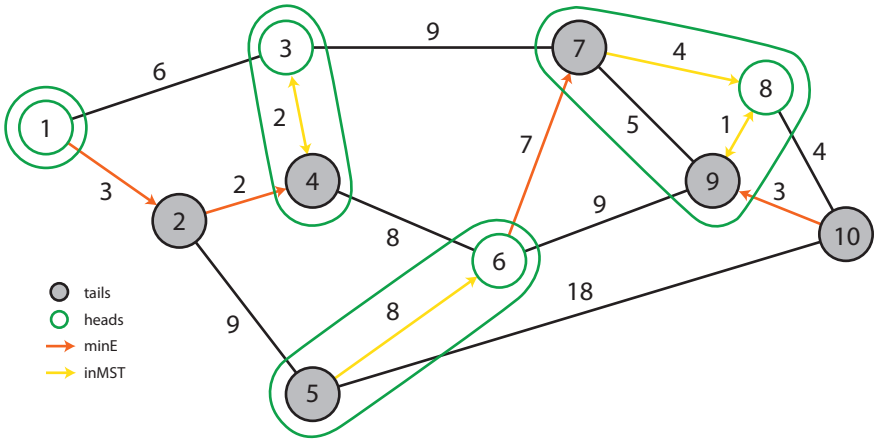
Q: What is the work and span of Prim's algorithm?

A: $O(m \log n)$ (same as Dijkstra's).

2 MST

In lecture 17 we presented Boruvka's algorithm, a parallel algorithm for finding a MST. The idea is similar to star contraction, but instead of contracting any of the edges, we 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:



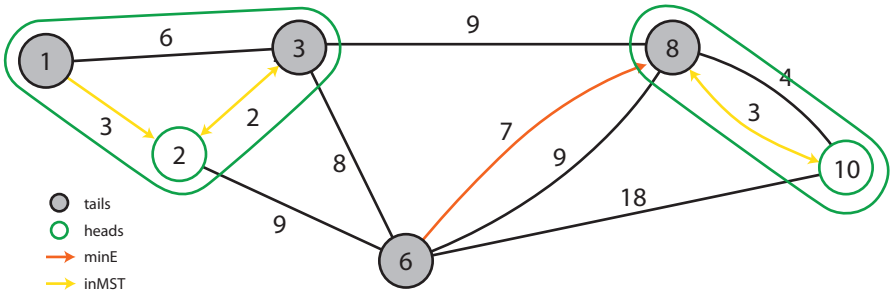
In the first round of the algorithm, we have the following flips:

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.

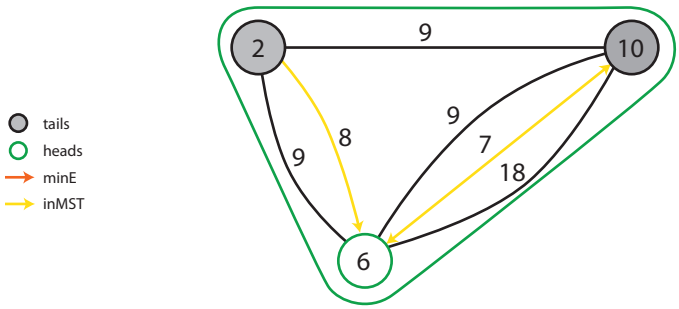
We get the following contracted graph in the next round:



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 expectations, $n/4$ vertices in expectation will be removed in each round.

3 Linked List Scan

We have a linked list of nodes; each node i has some data D_i. For each node, we want to compute $T_i = \sum_{\text{j after i in M}} D_j$. Let $S[i]$ be the index of the successor of node i. If there is no successor then $S[i] = i$. That is, S is a permutation of {0, 1, . . . , n-1} except that the last node points to itself. This problem is known as *list ranking*.

This looks like the familiar scan problem on sequences except that it computes the sum of the suffixes. But it is harder than scan: we don't know where in the list each element is. That is, the i^{th} node in the list is not necessarily at position i.

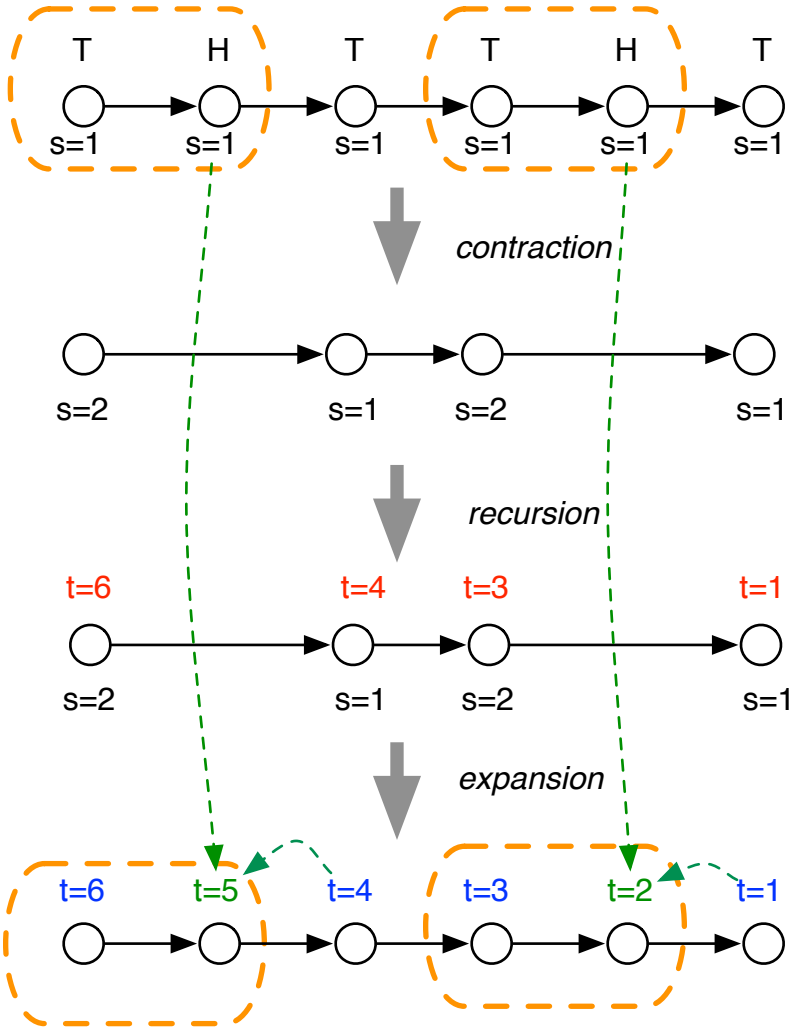
But we can use the same idea: combine pairs of adjacent nodes, recurse on the smaller list, and then expand back the list. With sequences we paired elements at even positions with the following odd elements. In linked lists, we don't know which nodes are even and which are odd.

Q: How do we stop a node from ending up in two pairs?
A: Restrict each node to being paired with only its predecessor node or only its successor node. Then each node can only be part of one pair.

Q: How do we mark the nodes? We don't have much to go on...
A: Try random (random is usually worth a try).

Each node flips a fair coin. If the flip comes up tails then it is an "even" node; if it comes up heads then it is an "odd" node. We pair only ("even","odd") adjacent nodes. That is, a node is paired with the next node if it flips tails and its successor flips heads. What is the probability that node i is paired with its successor in the list? This happens with probability 1/4 (we must have marked i with tails and $S[i]$ with heads, each of which occurs indepedently with probability 1/2). By linearity of expectations, we get $(n-1)/4$ pairs total in expectation (the last node can't start a pair).

Now it's just details: To contract a pair we *splice out* the second node from each pair from the list. Let (x, y) be a pair. We are deleting y from the list, so we need to make the successor to x be y's successor, that is $S'[x] = S[y]$. Now we need to add in y's data to x's data: $D'[x] = D[x] + D[y]$. Then recurse on the new list (the old list minus all the y's). The result returned is, everything that wasn't a y has the correct sum T (by IH). For each y, add in the result of its successor: $T[y] = D[y] + T[S[y]]$.



In the code below, I is sequence of indices of nodes that are still in the linked list, X are nodes that are the first of a pair, and Y are the second of a pair.

Let's do an example: (keep track of S, D, I at each step; I is the set of active indices)
$S = \{2, 0, 4, NONE, 3\}$
$D = \{2, 3, 5, 1, 4\}$
$I = \{0, 1, 2, 3, 4\}$

Let's say we get $TTHHT$ on our first set of flips. Show the steps for the first round of the algorithm, do the recursive call, and then show how to reconstruct the list.

Answer:
$(0, 2)$ and $(4, 3)$ pair.

So we splice out $2, 3$ and get:

$S' = \{4, 0, 4, NONE, NONE\}$
$D' = \{7, 3, 5, 1, 5\}$
$I = \{0, 1, 4\}$

Now we recursively compute $T[0] = 12, T[1] = 15, T[4] = 5$
Then we can compute $T[2] = D[2] + T[4] = 10$ and $T[3] = D[3] + 0 = 1$.