Recitation 10 — Minimum Spanning Trees

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

October 30, 2013

1 Announcements

- Assignment 7 is due on Monday.
- Today's recitation will review MSTs (important for the assignment) and do a practice problem, similar to something that might be on an exam.

2 Minimum Spanning Trees

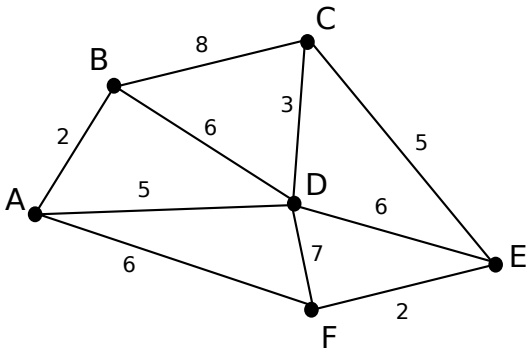
Recall that a minimum spanning tree (MST) is a spanning tree of a weighted graph, such that the sum of the edge weights in the tree is as small as possible. Today, we will review two algorithms for finding an MST of a graph. Prim's algorithm is a simple, sequential algorithm similar to algorithms like Dijkstra and A^*, and Boruvka's is a parallel algorithm based on graph contraction.

2.1 Prim's Algorithm

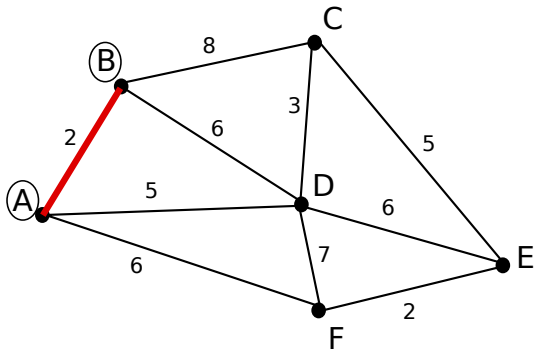
Algorithm:

1. Choose any starting vertex. Look at all edges connecting to the vertex. Choose one of the ones with the lowest weight and add this to the tree.
2. Look at all edges with exactly one endpoint 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.

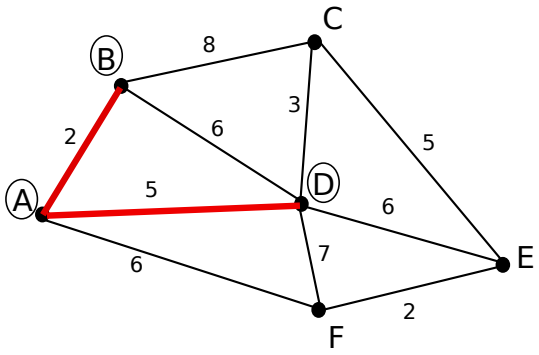
Let's run through Prim's algorithm on this example graph.



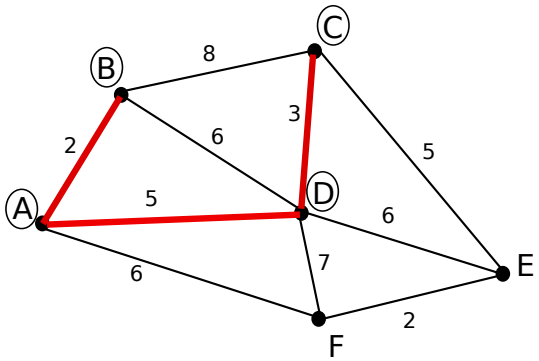
- Choose vertex A. Choose edge with lowest weight: (A,B).



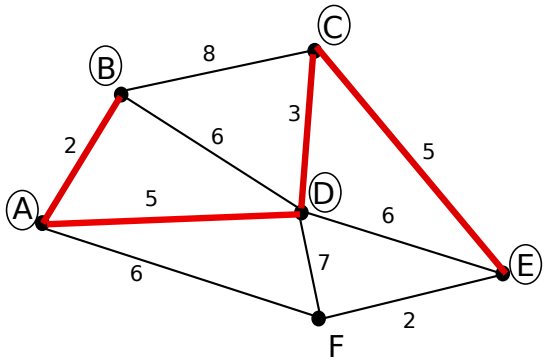
- Look at all edges connected to A and B: (B,C), (B,D), (A,D), (A,F). Choose the one with minimum weight and add it to the tree: (A,D).



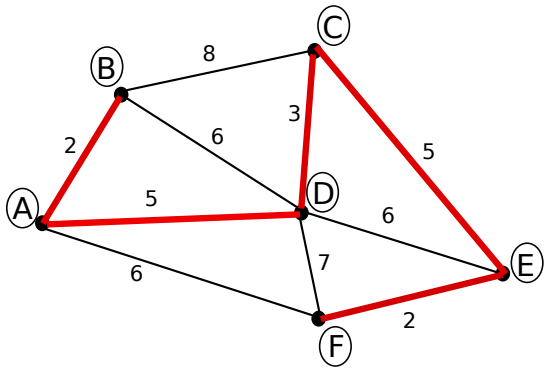
- Look at all edges connected to the tree: (B,C), (D,C), (D,E), (D,F), (A,F). We don't need to consider edge (B,D) because both B and D are in the tree already. We choose edge (D,C).



- 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: How do we know that Prim's algorithm will find an MST?

A: The light edge rule:

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

Theorem 2.1. *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.*

At every step, let U be the set of vertices currently in the tree. We always add the shortest edge between U and $V \setminus U$, which the light edge rule says is in the MST.

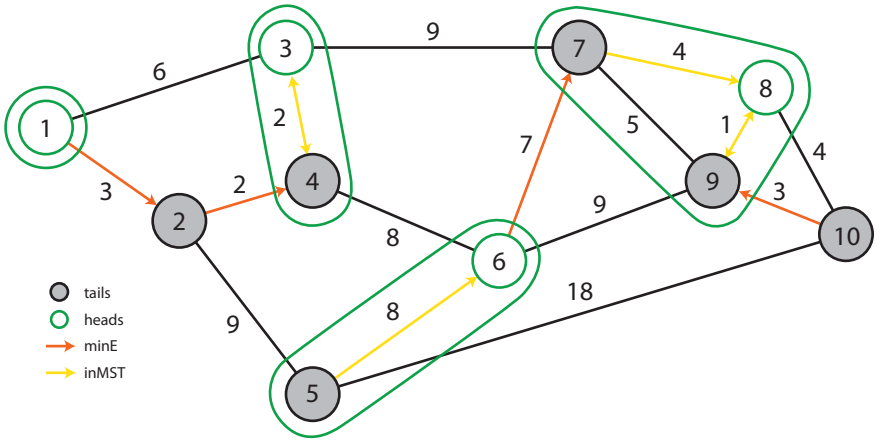
Q: What are the work and span of Prim's algorithm?

A: $O(m \log n)$ for both work and span (same as Dijkstra's).

2.2 Boruvka's Algorithm

In lecture, 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. Refer to the lecture notes for the pseudocode of this algorithm.

Let's go over an example:



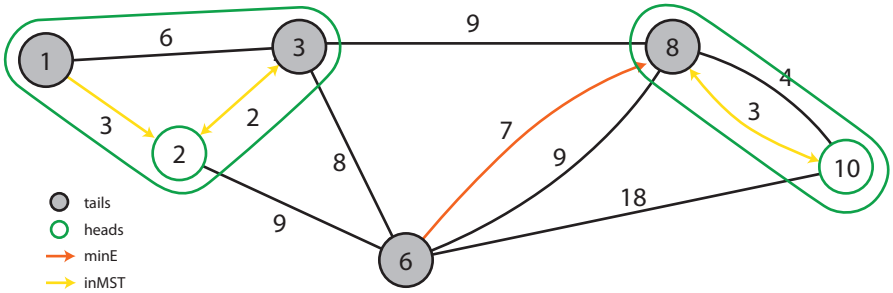
In the first round of the algorithm, we have the following flips:

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. Since the graph is undirected, the edge between 1 and 2 can be seen as an edge in both directions, but it is only a minimum out edge for 1, since the edge between 2 and 4 is lighter. Since we only contract minimum out edges that go from tails to heads, we do not contract 1 and 2.

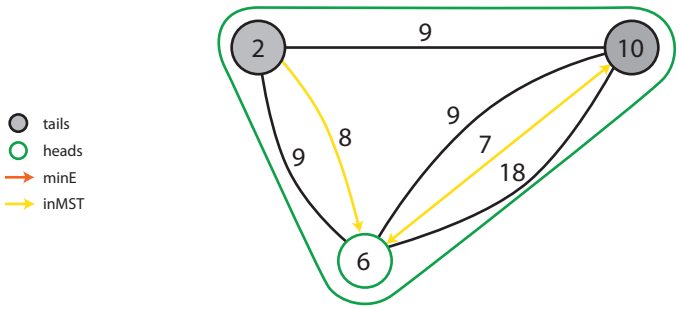
We get the following contracted graph in the next round:



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 out-edge which contracts with probability $1/4$. By linearity of expectations, $n/4$ vertices in expectation will be removed in each round.

3 Practice Problem

You are given a university course catalog with n courses. Every course has at most one prerequisite. Students at this university are not allowed to take a course without taking its prerequisite, if it has one, in a previous semester. Every course is offerred once every year, in either the fall or spring semester (assume that a course will be offerred either every fall or every spring but not both.) You wonder if there are any courses which it is impossible to take, because its prerequisites can't be met in eight semesters. Give an algorithm to determine if this is the case, in $O(n)$ work and $O(\lg^2 n)$ span.

3.1 Solution

Let's go through steps you might take to solve this problem if you don't immediately see the answer.

Note that by no means do you have to show all of this work or answer these questions if you're solving a problem on a test. But, if you're stuck, thinking through these steps could help you come up with an answer.

1. *State the problem formally, in terms of data structures and concepts we've seen in class.*

   This problem, like many you may encounter, can be viewed as a graph problem. The red flags of a graph problem in this case are items which have dependencies on each other (i.e. vertices and directed edges). Make a graph in which courses are the vertices and each vertex has an edge toward all of the courses for which it is a prequisite (you could also make the edges point the other way, but might change your mind later).

2. *Encode the remaining features of the problem and the question in your representation (this may cause you to rethink your representation.)*

If edges point from prerequisites to classes, then following an edge is like taking a class after having taken its prerequisite. This is why we had edges pointing from, not toward, prerequisites. If you had made your graph the other way, you might decide to change it at this point. So a traversal of a graph is like a semester-by-semester course plan.

Q: What does it mean that every course has at most one prerequisite?
A: Think of what a connected component of this graph would look like: it would either be a tree with the root being a course with no prerequisite, or it is a cycle of courses that all depend on each other.

Q: What does it mean that a course can be offerred in the spring or fall?
A: After taking a fall class, you can take a spring class one semester later, but must wait two semesters to take another fall class (and similar for spring classes). Since edges correspond to taking classes, you can encode how long you have to wait to take the next class as an edge weight. Edges from fall to fall or spring to spring should have weight 2 and from fall to spring or spring to fall should have weight 1.

Q: What does it mean for it to be impossible to take a course?
A: We will take the course if we reach it on a traversal of the graph. Only having 8 semesters means we can only traverse paths of length 7 from a root that is a fall class, and paths of length 6 from a root that is a spring class. Any class that cannot be reached by a traversal of this kind (including any that is part of a cycle) cannot be taken.

3. *Match the statement of the problem, encoded in your representation, to an algorithm you know that solves a similar problem. The cost bounds can help you. This may require modifying your encoding again.*

   Any graph search, such as DFS or BFS, will do to solve the problem. Look at the cost bounds though: is there a graph search that runs in those cost bounds? BFS with tables and sets runs in $O(m)$ work and $O(d \lg^2 n)$ span. Remember though that, since every course has at most one prerequisite, $m \leq n$, so $O(m) = O(n)$. What about d? This is the maximum search depth. However, we don't want to search past depth 8, since at this point we can regard the rest of the classes as unreachable. Thus, d is a constant and $O(d \lg^2 n) = O(\lg^2 n)$. This gives the cost bounds we need. Since unmodified BFS doesn't work on weighted graphs, we can introduce a "dummy" vertex along each edge of weight 2, making it into two edges of weight 1.

4. *Write out your algorithm in pseudocode or clear prose.*

   The function `canTake` takes the course catalog represented as a directed, unweighted graph as described above (with dummy nodes between courses offerred in the same semester) and returns the set of classes that can be taken in 8 semesters. If $\texttt{canTake}(V, E) = V$, then all classes may be taken. Otherwise, $V \setminus \texttt{canTake}(V, E)$ contains the classes that can't be taken.

```
1  function canTake(V, E) =
2  let   function canTake'(X, F, d) =
3        if |F| = 0 ∨ d = 0 then X else
4        let
5            X' = X ∪ {v ↦ d : v ∈ F}
6            F' = N_G(F) \ domain(X')
7        in canTake'(X', F', d − 1)
8        end
9     function startDepth(v) =
10       if isFallClass(v) then 7 else 6
11 in
12    ⋃⟨canTake'({}, {v}, startDepth(v)) : v ∈ V, hasNoPrereqs(v)⟩
13 end
```

Note that we traverse starting at all classes with no prerequisites.