Recitation 11 – *Minimum Spanning Trees*

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2014)

March 25th, 2014

1 Announcements

- *SegmentLab* is out. **START EARLY!** This lab is one of the more complicated labs.
- *Friendly warning*: Exam 2 is next Friday, April 4th. :-)
- Any questions from lecture?

2 Prim's is the Easy One

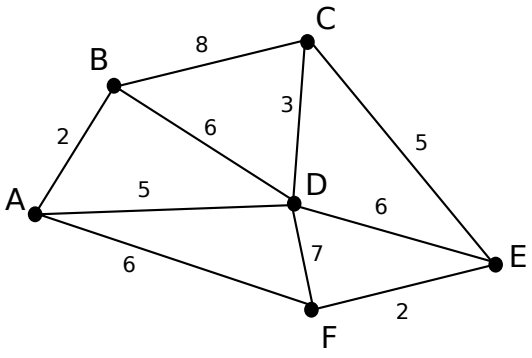
Q: What is a Minimum Spanning Tree?

A: An 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. Prim's algorithm is a simple, sequential algorithm similar to algorithms like Dijkstra and A^*, and Borůvka's is a parallel algorithm based on graph contraction.

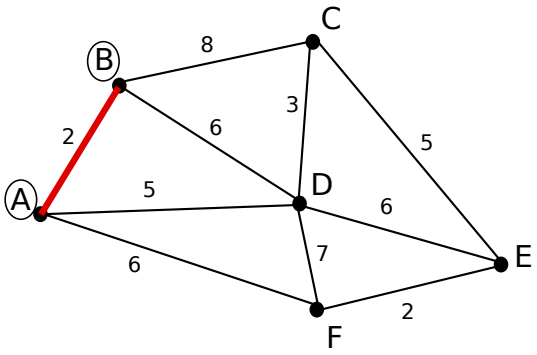
2.1 What exactly is the algorithm?

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.

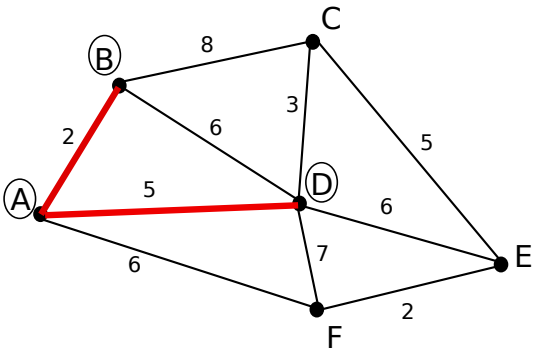
2.2 Can I see an example please?



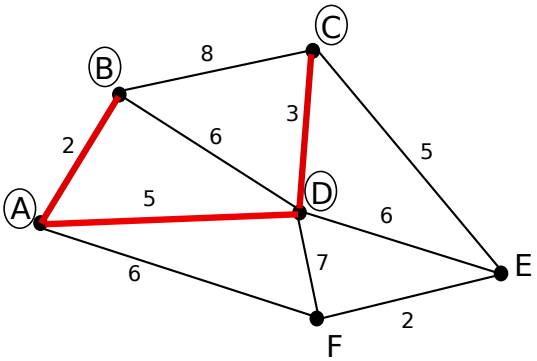
- Choose vertex A. Choose edge with lowest weight: (A,B).



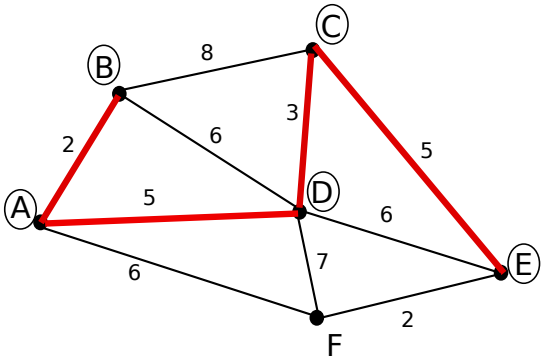
- Look at all edges connected to A and B: (B,C), (B,D), (A,D), (A,F). Choose the one with minimum weight and add it to the tree: (A,D).



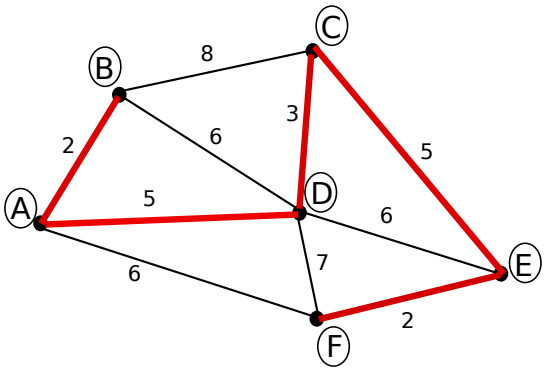
- Look at all edges connected to the tree: (B,C), (D,C), (D,E), (D,F), (A,F). We don't need to consider edge (B,D) because both B and D are in the tree already. We choose edge (D,C).



- 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.

2.3 Some important properties

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

A: The light edge rule states that **the lightest edge across a cut is in the MST of** G:

> **The Light Edge Rule**: 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).

3 Borůvka's Big Bold Idea

3.1 What is the algorithm?

Q: What is Borůvka's algorithm?

A: The idea behind Borůvka's algorithm is similar to star contraction, but instead of contracting any of the edges, we only contract the edges of minimum weight at each vertex.

Q: Why does this work?

A: The Light Edge Rule!

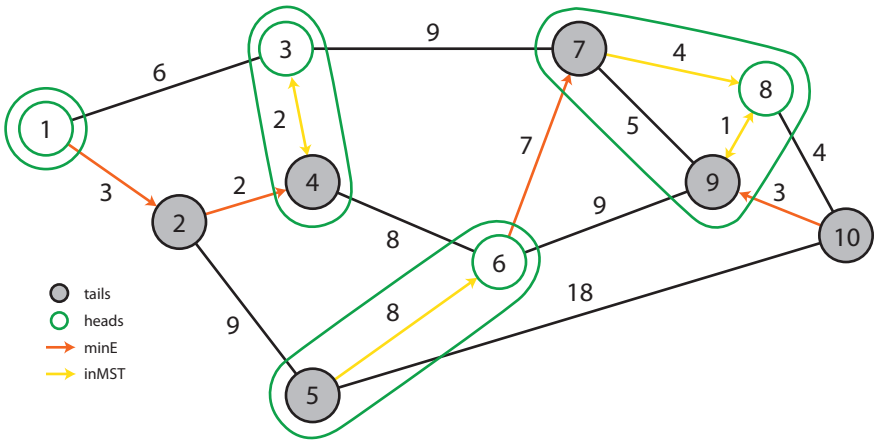
3.2 Can I see an example please?

Round 1

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

We contract the original graph as follows:



Q: Why are vertices 3 and 4 contracted, but 1 and 2 not?

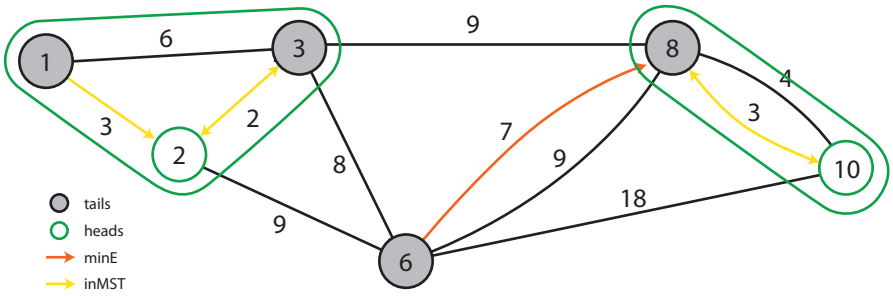
A: In our version of the algorithm, we only consider the minimum *out*-edges from every vertex. Since the graph is undirected, the edge between 1 and 2 can be seen as an edge in both directions, but it is only a minimum out-edge for 1, since the edge between 2 and 4 is lighter. Since we only contract minimum out edges that go from tails to heads, we do not contract 1 and 2.

Round 2

Here is the sequence of flips generated for the second round. **Note:** Although we generate another sequence of 10 flips here, we only look at the ones generated for the vertices which remain in our contracted graph.

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

We get the following contracted graph:

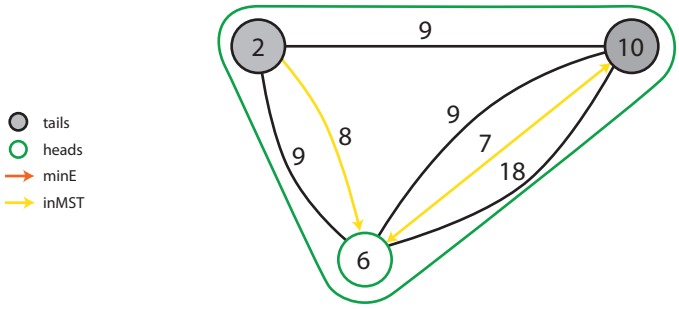


Round 3

Lastly, here are our flips for Round 3:

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

Which give us the final contraction:



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.

Q: How many vertices in expectation will be removed in each round?

A: Recall from lecture that each vertex has a minimum out-edge which contracts with probability $\frac{1}{4}$. By linearity of expectations, $\frac{n}{4}$ vertices in expectation will be removed in each round.

4 Practice Makes *Purr*-fect

4.1 Problem

You are given a university course catalog with n courses. Every course has *at most* one prerequisite, and a course cannot be taken concurrently with its prerequisite. 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 would be impossible to take because its prerequisites can't be met in eight semesters. Give an $O(n)$ work and $O(\log^2 n)$ span algorithm for this.

4.2 Solution

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.

4.2.1 State the problem formally

This problem 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).

4.2.2 Encode the features of the problem in 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.

4.2.3 Match the statement of the problem to a known algorithm

Any graph search, such as DFS or BFS, will solve the problem. We must, however, consider the cost bounds. BFS with tables and sets runs in $O(m)$ work and $O(d \log^2 n)$ 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 $O(d \log^2 n) = O(\log^2 n)$. 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.

4.2.4 Write out the 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. **Note:** we traverse starting at all classes with no prerequisites.

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

5 What Is This I Don't Even...

This pseudocode implements Borůvka's algorithm. **Make sure you fully understand what each individual line is doing**; this is a good jumping-off point for *SegmentLab*.

> **READ:** Grab a TA by their shirt (or buy them a nice drink!) and make them explain it to you *line-by-line* if you don't get it! This pseudocode has the ability to **blow minds**.

Fair warning: Don't just try to directly translate this into SML. I guarantee you that will be way more difficult than actually understanding it, digesting it, then putting on an *SML thinking cap* and rethinking the algorithm.

Here we go... Given that `minEdges` finds the minimum edge out of each vertex in the graph, here is `minStarContract`:

```
1  fun minStarContract(G = (V, E), i) =
2    let
3      val minE = minEdges(G)
4      val P = {u ↦ (v, w, l) ∈ minE | ¬heads(u, i) ∧ heads(v, i)}
5      val V' = V \ domain(P)
6    in
7      (V', P)
8    end
```

And here is `MST`:

```
1  fun MST((V, E), T, i) =
2    if |E| = 0 then T
3    else let
4      val (V', PT) = minStarContract((V, E), i)
5      val P = {u ↦ v : u ↦ (v, w, ℓ) ∈ PT} ∪ {v ↦ v : v ∈ V'}
6      val T' = {ℓ : u ↦ (v, w, ℓ) ∈ PT}
7      val E' = {(P[u], P[v], w, l) : (u, v, w, l) ∈ E | P[u] ≠ P[v]}
8    in
9      MST((V', E'), T ∪ T', i + 1)
10   end
```

Good luck on *SegmentLab*! :-)