Design and Analysis of Algorithms
Massachusetts Institute of Technology
Debayan Gupta, Aleksander Madry, Bruce Tidor

March 2, 2018
6.046/18.410
Recitation 4

Recitation 4: Max Flow, MST

1 Minimum Spanning Trees

Definition 1 *A **spanning tree** T of a graph G is an acyclic subset of G's edges that connects all the vertices of G.*

The Minimum Spanning Tree Problem: Given a connected, undirected graph $G = (V, E, w)$ with edge weights $w(\{u, v\})$, find the spanning tree of G of minimum weight, where the weight of a tree T is defined as $w(T) = \sum_{\{u,v\} \in T} w(\{u, v\})$.

Definition 2 *Imagine growing an MST one edge at a time. At each step, we would like to make sure that the edge we are adding will produce a new set of edges that is still a subset of some MST. We will call edges that satisfy this property **safe** edges.*

Here's what the above algorithm looks like in pseudocode:

GENERIC-MST$(G = (V, E, w))$

```
1: T = ∅
2: while T is not spanning do
3:     Find an edge {u, v} that is safe for T
4:     Add {u, v} to T
5: end while
6: return T.
```

The algorithm maintains the invariant that (while T is not yet spanning) a *safe* edge for T must exist. The hard part is finding such a safe edge.

We will first prove some general properties about the MST problem before showing how they relate to specific algorithms.

Greedy Choice Property: Intuitively, this property means that we can build up a globally optimal solution by making locally optimal choices. In particular, for MSTs, we can make the following claim:

*Let S be some non-empty proper subset of the vertices. For any cut $(S, V \setminus S)$, any least-weight edge $e = \{u, v\}$ with $u \in S, v \notin S$ (i.e. a "crossing" edge) must belong to some MST. This is also known as the **cut property**.*

Proof: We can prove this property using an exchange argument. Consider any least-weight crossing edge $e = \{u, v\}$ for some cut $(S, V \setminus S)$. Consider any MST T. If T contains e, we're done. Suppose (u, v) is not in any MST. Then, we can consider the unique path from u to v in T, which

must exist since T is a spanning tree. We can swap (u, v) with the first edge on this path that connects a vertex in S to a vertex in $V \setminus S$. Since (u, v) is the lightest edge across this cut, the new tree, T' that we have produced a has at most the weight of the original MST and is therefore also an MST and e belongs to some MST, as desired. □

Next, we introduce the idea of a graph contraction.

Definition 3 *A **contraction** of a graph G on an edge $e = \{u, v\}$ produces a graph G/e where u and v are merged into a single node uv (destroying edge e), and all other edges involving either u or v have uv as their endpoint instead.*

Optimal Substructure Property: Intuitively, this property means that the optimal solution to the entire problem, contains within it the optimal solution to subproblems. For MSTs, this property translates to the following claim:

If an edge e belongs to some MST T^* of G, and T' is some MST of $G' = G/e$, then $T = T' \bigcup \{e\}$ is an MST of G.

Proof: First, notice that T is a spanning tree of G and that T^*/e is a spanning tree of G'. Since T' is an MST of G', we have $w(T') \leq w(T^*/e)$. Therefore, we get the following chain of inequalities:

$$w(T) = w(T') + w(e) \leq w(T^*/e) + w(e) = w(T^*)$$

Thus T is an MST of G. □

We suggest proving the following generalization to the above property, as an exercise: *If edges $e_1, e_2, ..., e_k$ belong to some MST of G, and T' is some MST of $G/\{e_1, ..., e_k\}$, then $T' \bigcup \{e_1, ..., e_k\}$ is an MST of G.*

Prim's Algorithm

Intuitively, *Prim's algorithm* grows a tree by greedily selecting the lowest weight edge that connects the tree to a vertex not in the tree.

PRIM$(G = (V, E, w))$

1: Pick starting vertex $s \in V$
2: $T_V = s$
3: $T_E = \varnothing$
4: **while** T is not spanning **do**
5: Find the lowest weight edge $e = (u, v)$ such that $u \in T_V, v \notin T_V$
6: Add v to T_V
7: Add e to T_E
8: **end while**
9: **return** The tree represented by T_V and T_E.

Correctness: We prove correctness by showing that at every iteration of the while loop, T_E is a subset of some MST of G, that is, this algorithm always adds safe edges.

To see this, consider the k^{th} iteration of the algorithm: let the set of edges already added to T_E be $e_1, e_2, ..., e_{k-1}$ and let $G' = G/\{e_1, ..., e_{k-1}\}$. Suppose that $e_k = (u, v)$ is the lowest-weight edge in G' such that $u \in T_V$ and $v \notin T_V$. Therefore, by the **cut property**, it must belong to some MST of G'; call this MST T'. By assumption, $e_1, ..., e_{k-1}$ are in some MST of G, so we can apply the **optimal substructure generalization** to show that $T' \bigcup \{e_1, ..., e_{k-1}\}$ is an MST of G. Thus the loop invariant holds. □

Runtime analysis: Since we need to connect all vertices, the outer loop will run $|V|$ times. Each repetition might require $O(|E|)$ work to examine all edges, for a total runtime of $O(|V||E|)$.

We can improve this runtime by not having to examine up to $O(|E|)$ edges when deciding which edge to add. By checking the edges leaving a vertex when we add that vertex to our tree, and remembering the lowest weight connection, we can reduce this runtime to $O(|E| + |V| \log |V|)$. Let's see how:

PRIM$(G = (V, E, w))$

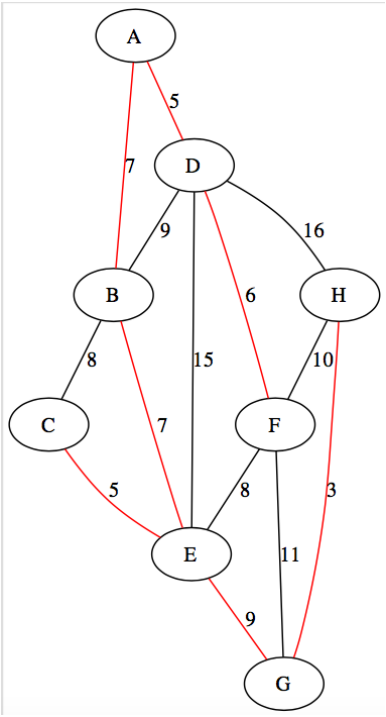
```
1:  T = ∅
2:  Create a priority queue Q on the vertices of G
3:  Pick starting vertex s ∈ V
4:  s.key = 0; for all other vertices, v.key = ∞
5:  while Q is not empty do
6:      u = EXTRACT-MIN(Q)
7:      add u to T
8:      for each neighbor v of u do
9:          if v ∈ Q and w(u, v) < v.key then
10:             # u is (so far) the node in the tree that is closest to v
11:             v.key = w(u, v)
12:             v.parent = u
13:         end if
14:     end for
15: end while
16: return {(v, v.parent) | v ∈ V \ s}
```

Observe that each edge is only examined once, so the total cost of the work done by the inner loop is $O(|E|) \cdot T_{Decrease-Key}$, where $T_{Decrease-Key}$ is the cost of decreasing the key of a member of the priority queue Q. Additionally, each iteration of the outer loop invokes **EXTRACT-MIN**, which has a cost of $T_{Extract-Min}$. Thus the total runtime is

$$O(|E| \cdot T_{Decrease-Key} + |V| \cdot T_{Extract-Min})$$

If we implement our priority queue using a Fibonacci heap, we get runtimes of $T_{Decrease-Key} = O(1)$ and $T_{Extract-Min} = O(\log |V|$, giving an overall runtime of $O(|E| + |V| \log |V|)$.

Example: Consider the following weighted, undirected graph. Use Prim's algorithm, initialized at the vertex v_A, to construct a minimum spanning tree for this graph.



We will grow the tree $T = (T_V, T_E)$, according to Prim's algorithm.

1. Initialize $T_V = \{A\}$ and $T_E = \{\}$.
2. Add the edge (A, D) to T_E and the vertex D to T_V.
3. Add the edge (D, F) to T_E and the vertex F to T_V.
4. Add the edge (A, B) to T_E and the vertex B to T_V.
5. Add the edge (B, E) to T_E and the vertex E to T_V.
6. Add the edge (C, E) to T_E and the vertex C to T_V.
7. Add the edge (E, G) to T_E and the vertex G to T_V.
8. Add the edge (G, H) to T_E and the vertex H to T_V.
9. $T = (T_V, T_E)$ is now a minimum spanning tree of G. Note that the picture of G has the edges from T_E colored red.

Kruskal's Algorithm

Kruskal's algorithm grows a forest, initially containing each vertex in a separate tree, by choosing minimum-weight edges to connect disjoint trees.

KRUSKAL$(G = (V, E, w))$

```
1:  T = ∅
2:  for v ∈ V do
3:      MAKESET(v)
4:  end for
5:  Sort all edges in E by weight in increasing order
6:  for e = (u, v) in E (in sorted order): do
7:      if FINDSET(u) ≠ FINDSET(v) then
8:          T = T ⋃ e
9:          UNION(u, v)
10:     end if
11: end for
12: return T
```

Line 7 guarantees that the graph returned will not have cycles. Because we iterate through all edges, we know that T will span the graph.[1] So T is a spanning tree, and we can prove that it is of minimal weight using the two generic MST properties established before. The proof is essentially the same as for Prim's, so we leave it as an exercise to fill in the details.

We conclude with another nice property of MSTs.

Cycle Property: *Consider any cycle C in the graph G. If C contains an edge e whose weight is strictly larger than the weight of every other edge in the cycle, then e cannot belong to any MST.*

Proof: We can prove this by contradiction. Suppose that $e = \{u, v\}$ does actually belong to some MST T. If we remove e, this will produce two disjoint trees, call them T_u and T_v after which endpoint of e they contain. Now, imagine walking the rest of the cycle from u to v. At some point we will traverse an edge that crosses from T_u to T_v. Adding this edge back would produce a spanning tree, and because we assumed e is strictly heavier than every other edge in the cycle, the resulting spanning tree would have a lower total weight than the original tree. Thus T is not an MST, and this is a contradiction. □

[1] Suppose T does not span the graph. Because G is connected, there exists a path in G between disconnected components of T. At least one of the edges in that path would have passed the test on line 7 and would been added to T.

2 Network Flows

Definition 4 *A **flow network** is a directed graph $G = (V, E)$ where each edge (u, v) has a capacity $c(u, v) \geq 0$.*

We are interested in the flow from a source $s \in V$ to a target $t \in V$:

Definition 5 *A **flow** is a real-valued function that maps an edge $(u, v) \in E$ to a non-negative real number $f(u, v)$ such that:*
(a) f respects the capacity constraint: $f(u, v) \leq c(u, v)$ and
(b) the net flow is conserved (i.e. flow conservation*) - i.e., for all $u \in V - \{s, t\}$,*

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

*We define the **value** of a flow f, denoted $|f|$, as the difference in the flow out of the source and the flow into the source:*

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

We wish to know the maximum flow, which is formalized as the ***maximum-flow problem:*** *given a flow network G, with* source s *and* sink t*, find a flow that maximizes value.*

Definition 6 *A **cut** on a flow network $G = (V, E)$ is a partitioning of V into two non-empty subsets S and T such that S contains the source and T contains the target.*

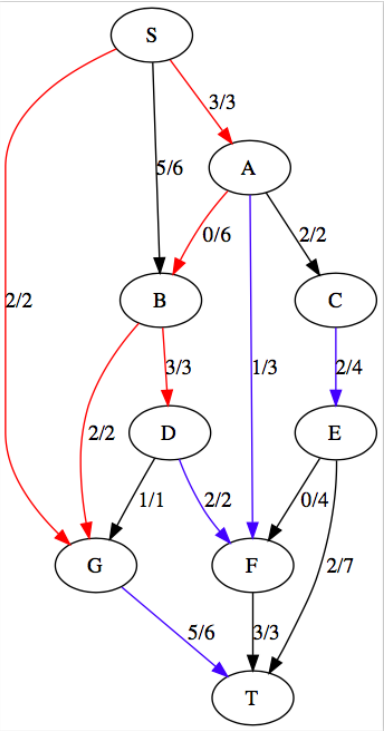
*Given a flow f, let the **net flow** $f(S, T)$ across the cut be defined as:*

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} (f(u, v) - f(v, u))$$

*and let the **capacity** of the cut be:*

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

Now let's take a look at an example at a flow network with two cuts across the network:



For each edge we have provided the flow value and capacity of that edge. The edges colored blue form one cut partitioning the source and drain nodes S and T, while the edges colored red form a different cut. Note that the net flow over the two cuts are the same: 10.

Suppose we are given a flow network G and a flow f, and we want to characterize the additional capacity of G to carry more flow. Since any new flow cannot exceed the original capacity constraint, let us define new capacities of G under the flow f. Namely, let $c_f(u, v) = c(u, v) - f(u, v)$.

Definition 7 *Given a graph $G = (V, E)$ and a flow f, the **residual network** is a graph G_f with vertices V and edges $E_f = \{(u, v) \in V \times V | c_f(u, v) > 0\}$.*

Definition 8 *Given a flow network and a flow, an **augmenting path** is a path in the residual network from the source to the target. Note that the **residual capacity** of an augmenting path is given by the minimum residual-capacity of any edge in the path.*

The Maximum-Flow Minimum-Cut Theorem: Given a flow network $G = (V, E)$ with source s and sink t. A flow f is a maximum flow in G if and only if the value of f is equal to the capacity of some minimal cut of G.

One can show that a residual graph G_f has no augmenting paths exactly when f is a max flow.