Design and Analysis of Algorithms

Massachusetts Institute of Technology — 6.046J/18.410J
Debayan Gupta, Aleksander Madry, and Bruce Tidor — Handout 4

Recitation 4: Greedy Algorithms & Network Flows

1 Minimum Spanning Trees

The Minimum Spanning Tree Problem: given a connected, undirected graph $G = (V, E, w)$ with edge weights $w(\{u, v\})$, find a set of edges $T \subseteq E$ that is acyclic, connects all of the vertices in G and minimizes the total weight $w(T) = \sum_{\{u,v\}\in T} w(\{u, v\})$.

Definition 1 *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.*

Now consider the following algorithm for computing the minimum spanning tree.

```
GENERIC-MST($G = (V, E, w)$)
1: $T = \varnothing$
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 $T$.
```

In the above algorithm, if T is not spanning, a *safe* edge for T must exist (otherwise the previously chosen edge was not in fact *safe*). The hard part, will be finding a *safe* edge.

We will first prove some general properties about the MST problem before showing how they relate to specific algorithms. First, there is the **greedy-choice property**, which says that the least weight edge crossing a cut belongs to some MST.

Greedy-choice property *For any cut $(S, V \setminus S)$ (with $S, V \setminus S \neq \emptyset$), any least-weight edge $e = \{u, v\}$ with $u \in S, v \notin S$ (i.e. a "crossing" edge) must belong to some MST.*

Proof: We can prove this property using an exchange argument. Suppose we have an MST T' that does not contain a least-weight crossing edge $e = \{u, v\}$ for the cut $(S, V \setminus S)$. It must contain some other edge $e' = \{u', v'\}$ that crosses the cut (otherwise T' is not spanning). Replacing e' with e produces a set of edges, call it T, whose weight is at most the weight of T' (by the assumption that e is a least-weight crossing edge). *So if we can show that T is still a spanning tree, we will have the desired contradiction and be done.*

Consider any node w. Since T' is an MST, w is connected to both u' and v'. Now, consider what happens when we remove e'. This splits T' into two connected components, one containing u' and one containin v'. w must be in one of these connected components; suppose WLOG it is in the one containing u'. But by a similar argument for e, u' must be connected to either u or v in $T' \setminus \{e'\}$. And this means that when we add in the edge e to create T, w is connected to both u and v. Thus T is still connected, and we can conclude that T is a spanning tree. □

Next, we introduce the idea of a graph contraction, which shall allow us to can state and prove the **optimal substructure property** for MSTs.

Definition 2 *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: *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: We note that T is a spanning tree and that T^*/e is a spanning tree of G' implying $w(T') \leq w(T^*/e)$, which in turn implies:

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

Thus T is an MST. □

Intuitively, the **optimal substructure property** allows us to build up optimal solutions from optimal solutions to subproblems. An essentially similar proof shows the following generalization: *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: $T_V = \varnothing$
2: $T_E = \varnothing$
3: while $T$ is not spanning do
4:     Find the lowest weight edge $e = (u, v)$ such that $u \in T_V, v \notin T_V$
5:     Add $v$ to $T_V$
6:     Add $e$ to $T_E$
7: end while
8: 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 - i.e., 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 $G' = G/\{e_1, ..., e_{k-1}\}$. Now, $e_k = (u, v)$ is the lowest-weight edges overall in G' such that $u \in T_V$ and $v \notin T_V$, so by the **greedy-choice property** it must belong to some MST of G' - call this MST T'. By hypothesis for the previous iteration, $e_1, ..., e_{k-1}$ are in some MST of G and 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:

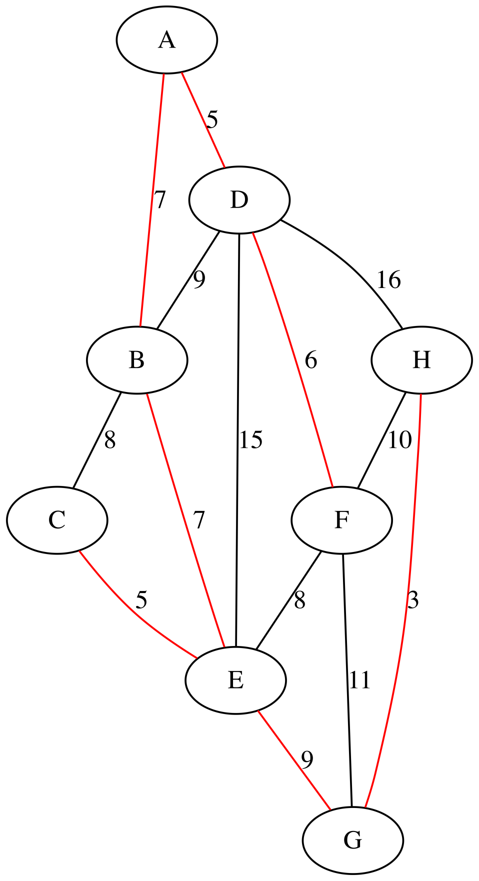
```
PRIM($G = (V, E, w)$)
1: $T = \varnothing$
2: Create a priority queue $Q$ on all vertices not in $T$
3: $s.key = 0$; for all other vertices, $v.key = \infty$
4: while $Q$ is not empty do
5:     $u$ = EXTRACT-MIN($Q$)
6:     add $u$ to $T$
7:     for each neighbor $v$ of $u$ do
8:         if $v \in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

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 $e_{A,D}$ to T_E and the vertex D to T_V.
3. Add the edge $e_{D,F}$ to T_E and the vertex F to T_V.
4. Add the edge $e_{A,B}$ to T_E and the vertex B to T_V.
5. Add the edge $e_{B,E}$ to T_E and the vertex E to T_V.
6. Add the edge $e_{C,E}$ to T_E and the vertex C to T_V.
7. Add the edge $e_{E,G}$ to T_E and the vertex G to T_V.
8. Add the edge $e_{G,H}$ 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: $T = \varnothing$
 2: for $v \in V$ 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($u$) $\neq$ FINDSET($v$) then
 8:       $T = T \bigcup e$
 9:       UNION($u, v$)
10:    end if
11: end for
12: return $T$
```

Line 7 guarantees that what we return, T, 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 the reader to fill in the details.

2 Network Flows

Definition 3 *A **flow network** is a directed graph $G = (V, E)$ where each edge has a non-negative capacity and the following property holds: for $u, v \in V$, if $(u, v) \in E$ then $(v, u) \notin E$.*

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

Definition 4 *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) $f(u, v)$ is bounded by the capacity of (u, v) (this is known as the* capacity constraint*) and (b) the net flow is conserved (i.e.* flow conservation*) - i.e., for all $u \in V - \{s, t\}$,*

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

[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.

*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 are naturally inclined to ask what is 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 5 *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.*

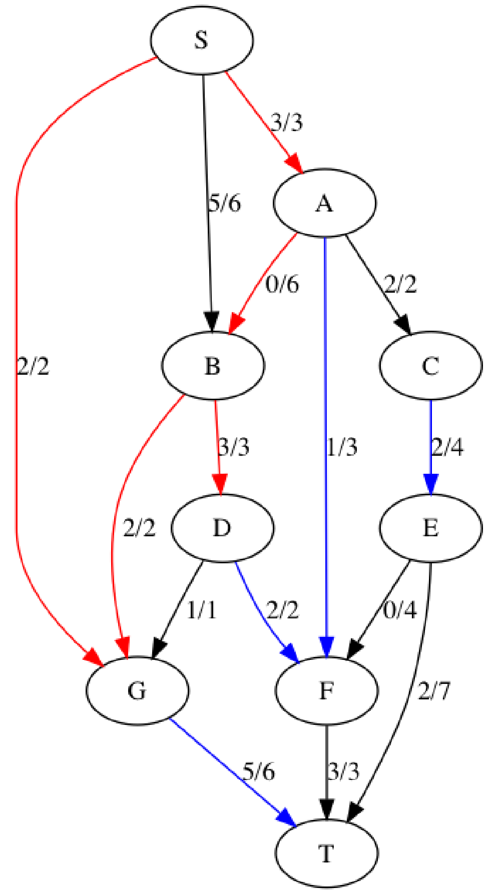
*Given a flow f, let the **net flow** $f(S,T)$ across the cut be defined as:*

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

*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.

Given a flow network and a flow, we formalize the notion of the remaining capacity in the network after we deduct the flow as follows:

Definition 6 *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 7 *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 and sink $s, t \in V$ and a flow f, f is a maximum flow in G if and only if: (a) the residual network does not have any augmenting paths and (b) the value of f is equal to the capacity of some minimal cut of G.

Intuitively, the max-flow min-cut theorem states that the value of the maximum flow is the equal to the capacity of a minimum cut of the flow network.

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 any 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. $\square$