VI Graph Algorithms

Elementary Graph Algorithms
Minimum Spanning Trees
Single-Source Shortest Paths
All-Pairs Shortest Paths

22 Elementary Graph Algorithms

- There are two standard ways to represent a graph $G = (V, E)$:
  - as a collection of adjacency lists or
  - as an adjacency matrix
- Either way applies to both directed and undirected graphs
- The adjacency-list representation provides a compact way to represent sparse graphs — those for which $|E| \ll |V|^2$
23 Minimum Spanning Trees

- Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.
- To interconnect a set of $n$ pins, we can use an arrangement of $n - 1$ wires, each connecting two pins.
- Of all such arrangements, the one that uses the least amount of wire is usually the most desirable.

Model this wiring problem with a connected, undirected graph $G = (V,E)$, where
- $V$ is the set of pins,
- $E$ is the set of possible interconnections between pairs of pins, and
- for each edge $(u,v) \in E$, we have a weight $w(u,v)$ specifying the cost (amount of wire) to connect $u$ and $v$.

Find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight is minimized

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$
Since $T$ is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it “spans” the graph $G$.

We call the problem of determining the tree $T$ the minimum-spanning-tree problem (MST).

We examine two algorithms for solving the MST problem: Kruskal’s and Prim’s algorithms.

We can easily make each of them run in time $O(E \lg V)$ using ordinary binary heaps.

By using Fibonacci heaps, Prim’s algorithm runs in time $O(E + V \lg V)$, which improves over the binary-heap implementation if $|V| \ll |E|$.

The two algorithms are greedy algorithms.

Greedy strategy does not generally guarantee finding globally optimal solutions to problems.

For the MST problem we can prove that greedy strategies do yield a tree with minimum weight.
23.1 Growing a minimum spanning tree

- Assume that we have a connected, undirected graph $G = (V,E)$ with a weight function $w: E \to \mathbb{R}$, and we wish to find a MST for $G$
- The following generic method grows the MST one edge at a time
- The generic method manages a set of edges $A$, maintaining the following loop invariant:
  
  Prior to each iteration, $A$ is a subset of some minimum spanning tree

At each step, we determine an edge $(u,v)$ that we can add to $A$ without violating this invariant, in the sense that $A \cup \{(u,v)\}$ is also a subset of a MST
- We call such an edge a safe edge for $A$

**Generic-MST**$(G,w)$

1. $A \leftarrow \emptyset$
2. while $A$ does not form a spanning tree
3. find an edge $(u,v)$ that is safe for $A$
4. $A \leftarrow A \cup \{(u,v)\}$
5. return $A$
**Initialization:** After line 1, the set $A$ trivially satisfies the loop invariant

**Maintenance:** The loop in lines 2–4 maintains the invariant by adding only safe edges

**Termination:** All edges added to $A$ are in a MST, and so the set $A$ returned in line 5 must be a MST

- The tricky part is finding a safe edge in line 3
- One must exist, since the invariant dictates that there is a spanning tree $T$ such that $A \subseteq T$
- Within the while loop body, $A$ must be a proper subset of $T$, and there must be an edge $(u,v) \in T$ s.t. $(u,v) \notin A$ and $(u,v)$ is safe for $A$

**Cut** $(S, V - S)$ of an undirected graph $G = (V,E)$ is a partition of $V$

- An edge $(u,v) \in E$ **crosses** the cut if one of its endpoints is in $S$ and the other is in $V - S$
- We say that a cut **respects** a set $A$ of edges if no edge in $A$ crosses the cut
- A **light edge** crossing a cut has the minimum weight of any edge crossing the cut
- Note that there can be ties
- More generally, an edge is a light edge satisfying a given property if its weight is the minimum of any edge satisfying the property
Theorem 23.1: Let

- \( G = (V, E) \) be a connected, undirected graph
- with a real-valued weight function \( w \) on \( E \).
- Let \( A \) be a subset of \( E \) that is included in some MST for \( G \),
- let \((S, V - S)\) be any cut of \( G \) that respects \( A \), and
- let \((u, v)\) be a light edge crossing \((S, V - S)\).

Then, edge \((u, v)\) is safe for \( A \).
Theorem 23.1 gives us a better understanding of the workings of the GENERIC-MST method on a connected graph $G = (V,E)$

As the method proceeds, the set $A$ is always acyclic; otherwise, a MST including $A$ would contain a cycle, which is a contradiction

At any point in the execution, the graph $G_A = (V,A)$ is a forest, and each of the connected components of $G_A$ is a tree

Some of the trees may contain just one vertex, as is the case, e.g., when the method begins: $A$ is empty and the forest contains $|V|$ trees, one for each vertex

Moreover, any safe edge $(u,v)$ for $A$ connects distinct components of $G_A$, since $A \cup \{(u,v)\}$ must be acyclic

The while loop in lines 2–4 of GENERIC-MST executes $|V| - 1$ times because it finds one of the $|V| - 1$ edges of a minimum spanning tree in each iteration

Initially, when $A = \emptyset$, there are $|V|$ trees in $G_A$, and each iteration reduces that number by 1

When the forest contains only a single tree, the method terminates
Corollary 23.2: Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function $w$ defined on $E$.

Let $A$ be a subset of $E$ that is included in some MST for $G$, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If $(u, v)$ is a light edge connecting $C$ to some other component in $G_A$, then $(u, v)$ is safe for $A$.

Proof: The cut $(V_C, V - V_C)$ respects $A$, and $(u, v)$ is a light edge for this cut. Therefore, $(u, v)$ is safe for $A$. 

23.2 The algorithms of Kruskal and Prim

- These algorithms use a specific rule to determine a safe edge in line 3 of GENERIC-MST
- In Kruskal’s algorithm, the set $A$ is a forest whose vertices are all those of the given graph
- The safe edge added to $A$ is always a least-weight edge in the graph that connects two distinct components
- In Prim’s algorithm, the set $A$ forms a single tree
- The safe edge added to $A$ is always a least-weight edge connecting the tree to a vertex not in the tree
**Kruskal’s algorithm**

- Find a safe edge to add to the growing forest by finding, of all the edges that connect any two trees in the forest, an edge \((u, v)\) of least weight
- Let \(C_1\) and \(C_2\) denote the two trees that are connected by \((u, v)\)
- Since \((u, v)\) must be a light edge connecting \(C_1\) to some other tree, Corollary 23.2 implies that \((u, v)\) is a safe edge for \(C_1\)
- This is a greedy algorithm because at each step it adds an edge of least possible weight

**MST-KRUSKAL\((G, w)\)**

1. \(A \leftarrow \emptyset\)
2. for each vertex \(v \in G.V\)
3. \(\text{MAKE-SET}(v)\)
4. sort the edges of \(G.E\) into nondecreasing order by weight \(w\)
5. for each edge \((u, v) \in G.E\), taken in nondecreasing order by weight
6. if \(\text{FIND-SET}(u) \neq \text{FIND-SET}(v)\)
7. \(A \leftarrow A \cup \{(u, v)\}\)
8. \(\text{UNION}(u, v)\)
9. return \(A\)
• **FIND-SET(\(u\))** returns a representative element from the set that contains \(u\)

• Determine whether \(u\) and \(v\) belong to the same tree by testing \(\text{FIND-SET}(u) = \text{FIND-SET}(v)\)

• **UNION** procedure combines trees

• Lines 1–3 initialize the set \(A\) to the empty set and create \(|V|\) trees, one containing each vertex

• The for loop in lines 5–8 examines edges in order of weight, from lowest to highest

• The loop checks, for each edge \((u, v)\), whether the endpoints \(u\) and \(v\) belong to the same tree

• If they do, then the edge \((u, v)\) cannot be added to the forest without creating a cycle, and the edge is discarded

• Otherwise, the two vertices belong to different trees

• In this case, line 7 adds the edge \((u, v)\) to \(A\), and line 8 merges the vertices in the two trees
The running time depends on how we implement the disjoint-set data structure. Initializing the set $A$ (line 1) takes $O(1)$ time, and the time to sort the edges (line 4) is $O(E \lg E)$. The for loop (lines 5–8) performs $O(E)$ FIND-SET and UNION operations on the disjoint-set forest. With the $|V|$ MAKE-SET operations, these take a total of $O((V + E)\alpha(V))$ time $\alpha$ is the very slowly growing function. We assume that $G$ is connected, so have $|E| \geq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(\lg V) = O(\lg E)$, the total running time of Kruskal’s algorithm is $O(E\lg E)$. Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg V)$, and the running time of Kruskal’s algorithm is $O(E\lg V)$.

**Prim’s algorithm**

- Prim’s algorithm has the property that the edges in the set $A$ always form a single tree.
- We start from an arbitrary root vertex $r$ and grow until the tree spans all the vertices in $V$.
- Each step adds to $A$ a light edge that connects $A$ to an isolated vertex (no edge of $A$ is incident).
- By Corollary 23.2, this rule adds only edges that are safe for $A$ and eventually $A$ forms a MST.
- Greedy: each step adds to the tree an edge that contributes the min amount to the tree’s weight.
In order to implement Prim’s algorithm efficiently, we need a fast way to select a new edge to add to the tree formed by the edges in \( A \).

In the pseudocode below, the connected graph \( G \) and the root \( r \) of the MST to be grown are inputs to the algorithm.

During execution of the algorithm, all vertices that are not in the tree reside in a min-priority queue \( Q \) based on a key attribute.

For each vertex \( v \), the attribute \( v.key \) is the minimum weight of any edge connecting \( v \) to a vertex in the tree; by convention \( v.key = \infty \) if there is no such edge.

Attribute \( v.\pi \) names the parent of \( v \) in the tree.

The algorithm implicitly maintains the set \( A \) from GENERIC-MST as
\[
A = \{(v, v.\pi) | v \in V - \{r\} - Q\}
\]

When the algorithm terminates, the min-priority queue \( Q \) is empty; the minimum spanning tree \( A \) for \( G \) is thus
\[
A = \{(v, v.\pi) | v \in V - \{r\}\}
\]
MST-PRIM(\(G,w,r\))
\begin{enumerate}
\item for each \(u \in G.V\)
\item \(u.key \leftarrow \infty\)
\item \(u.\pi \leftarrow \text{NIL}\)
\item \(r.key \leftarrow 0\)
\item \(Q \leftarrow G.V\)
\item while \(Q \neq \emptyset\)
\item \(u \leftarrow \text{EXTRACT-MIN}(Q)\)
\item for each \(v \in G.\text{Adj}[u]\)
\item if \(v \in Q\) and \(w(u,v) \lt v.key\)
\item \(v.\pi \leftarrow u\)
\item \(v.key \leftarrow w(u,v)\)
\end{enumerate}

• The algorithm maintains the following three-part loop invariant:
• Prior to each iteration of the while loop of lines 6–11,
  \begin{enumerate}
  \item \(A = \{(v,v.\pi) | v \in V - \{r\} - Q\}\)
  \item The vertices already placed into the minimum spanning tree are those in \(V - Q\)
  \item For all vertices \(v \in Q\), if \(v.\pi \neq \text{NIL}\), then \(v.key \lt \infty\) and \(v.key\) is the weight of a light edge \((v,v.\pi)\) connecting \(v\) to some vertex already placed into the MST
  \end{enumerate}
- Line 7 identifies a vertex \( u \in Q \) incident on a light edge that crosses the cut \( (V - Q, Q) \).
- Removing \( u \) from the set \( Q \) adds it to the set \( V - Q \) of vertices in the tree, thus adding \( (u, u.\pi) \) to \( A \).
- The for loop of lines 8–11 updates the key and \( \pi \) attributes of every vertex \( v \) adjacent to \( u \) but not in the tree, thereby maintaining the third part of the loop invariant.

- The total time for Prim’s algorithm is \( O(V \log V + E \log V) = O(E \log V) \), which is asympt. the same as for Kruskal’s algorithm.
- We can improve the asymptotic running time of Prim’s algorithm by using Fibonacci heaps.
- If a Fibonacci heap holds \(|V|\) elements, an EXTRACT-MIN operation takes \( O(\log V) \) amortized time and a DECREASE-KEY operation (to implement line 11) takes \( O(1) \) amortized time.
- Therefore, by using a Fibonacci heap for the min-priority queue \( Q \), the running time of Prim’s algorithm improves to \( O(E + V \log V) \).
24 Single-Source Shortest Paths

- In a **shortest-paths problem**, we are given a weighted, directed graph $G = (V, E)$, with weight function $w: E \rightarrow \mathbb{R}$ mapping edges to real-valued weights.
- The weight $w(p)$ of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:
  \[ w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \]

- We define the shortest-path weight $\delta(u, v)$ from $u$ to $v$ by $\delta(u, v) =
  \begin{cases} 
    \min \{ w(p) : u \xrightarrow{p} v \} & \text{if there is a path from } u \text{ to } v \\
    \infty & \text{otherwise}
  \end{cases}$
- A shortest path from vertex $u$ to vertex $v$ is then defined as any path $p$ with weight $w(p) = \delta(u, v)$
Optimal substructure of a shortest path

- Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.
- Recall that optimal substructure is one of the key indicators that dynamic programming and the greedy method might apply.
- The following lemma states the optimal-substructure property of shortest paths more precisely.

Lemma 24.1 (Subpaths of shortest paths are shortest paths)
Given a weighted, directed graph $G = (V, E)$, with weight function $w: E \rightarrow \mathbb{R}$, let $p = (v_0, v_1, \ldots, v_k)$ be a shortest path from vertex $v_0$ to vertex $v_k$ and, for any $i$ and $j$ such that $0 \leq i \leq j \leq k$, let $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ be the subpath from vertex $v_i$ to $v_j$. Then, $p_{ij}$ is a shortest path from $v_i$ to $v_j$.

Proof: If we decompose path $p$ into $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$, then we have that $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Now, assume that there is a path $p_{ij}'$ from $v_i$ to $v_j$ with weight $w(p_{ij}') < w(p_{ij})$.

Then, $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}'} v_j \xrightarrow{p_{jk}} v_k$ is a path from $v_0$ to $v_k$ whose weight $w(p_{0i}) + w(p_{ij}') + w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that $p$ is a shortest path from $v_0$ to $v_k$.  

Negative-weight edges

- If the graph $G = (V,E)$ contains no negative-weight cycles reachable from the source $s$, then for all $v \in V$, the shortest-path weight $\delta(s,v)$ is well defined, even if it has a negative value.
- If $G$ contains a negative-weight cycle reachable from $s$, shortest-path weights aren’t well defined.
- No path from $s$ to a vertex on the cycle can be a shortest path—we always find a path with lower weight by following the proposed “shortest” path and then traversing the negative-weight cycle.

- If there is a negative-weight cycle on some path from $s$ to $v$, we define $\delta(s,v) = -\infty$.
Cycles

- A shortest path cannot either contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight.

- If \( p = (v_0, v_1, \ldots, v_k) \) is a path and \( c = (v_i, v_{i+1}, \ldots, v_j) \) is a positive-weight cycle on this path (\( v_i = v_j \) and \( w(c) > 0 \)), then the path \( p' = (v_0, v_1, \ldots, v_i, v_{j+1}, \ldots, v_k) \) has weight \( w(p') = w(p) - w(c) < w(p) \), and so \( p \) cannot be a shortest path from \( v_0 \) to \( v_k \).

If there is a shortest path from a source vertex \( s \) to a destination vertex \( v \) that contains a 0-weight cycle, then there is another shortest path from \( s \) to \( v \) without this cycle.

- We can repeatedly remove 0-weight cycles from the path until the shortest path is cycle-free.
- Therefore, we can assume that when we are finding shortest paths, they have no cycles, i.e., they are simple paths.

- Any acyclic path in \( G = (V, E) \) contains at most \( |V| \) distinct vertices, and at most \( |V| - 1 \) edges.
- Thus, we can restrict our attention to shortest paths of at most \( |V| - 1 \) edges.
Relaxation

- The attribute $v.d$ is an upper bound on the weight of a shortest path from source $s$ to $v$
- We call $v.d$ a **shortest-path estimate**
- The following $\Theta(V)$-time procedure initializes the shortest-path estimates and predecessors ($\pi$):

  \begin{align*}
  \text{INITIALIZE-SINGLE-SOURCE}(G,s) \quad &\quad 1. \text{ for each vertex } v \in G.V \\
  &\quad 2. v.d \leftarrow \infty \\
  &\quad 3. v.\pi \leftarrow \text{NIL} \\
  &\quad 4. s.d \leftarrow 0
  \end{align*}

- The process of relaxing an edge $(u,v)$ consists of testing whether we can improve the shortest path to $v$ found so far by going through $u$ and, if so, updating $v.d$ and $v.\pi$
- A relaxation step may decrease the value of the shortest-path estimate $v.d$ and update $v$’s predecessor attribute $v.\pi$
- The following code performs a relaxation step on edge $(u,v)$ in $O(1)$ time:

  \begin{align*}
  \text{RELAX}(u, v, w) \quad &\quad 1. \text{ if } v.d > u.d + w(u,v) \\
  &\quad 2. v.d \leftarrow u.d + w(u,v) \\
  &\quad 3. v.\pi \leftarrow u
  \end{align*}
Properties of shortest paths and relaxation

- To prove the following algorithms correct, we appeal to several properties of shortest paths and relaxation:

**Triangle inequality** (Lemma 24.10)

For any edge $(u, v) \in E$, we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$

**Upper-bound property** (Lemma 24.11)

We always have $v. d \geq \delta(s, v)$ for all vertices $v \in V$, and once $v. d$ achieves the value $\delta(s, v)$, it never changes.
No-path property (Corollary 24.12)
If there is no path from \( s \) to \( v \), then we always have \( v.d = \delta(s,v) = \infty \)

Convergence property (Lemma 24.14)
If \( s \to u \to v \) is a shortest path in \( G \) for some \( u, v \in V \), and if \( u.d = \delta(s,u) \) at any time prior to relaxing edge \((u,v)\), then \( v.d = \delta(s,v) \) at all times afterward.

Path-relaxation property (Lemma 24.15)
If \( p = (v_0, v_1, \ldots, v_k) \) is a shortest path from \( s = v_0 \) to \( v_k \), and we relax the edges of \( p \) in the order \((v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)\), then \( v_k.d = \delta(s,v_k) \). This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of \( p \).

Predecessor-subgraph property (Lemma 24.17)
Once \( v.d = \delta(s,v) \) for all \( v \in V \), the predecessor subgraph is a shortest-paths tree rooted at \( s \).