Topic 11:
Shortest Path
Basics
Dijkstra Algorithm
Shortest Path

- Given a graph \( G = (V, E) \) with weight function \( w : E \rightarrow R \)

- **Weight of path** \( p = \langle v_0, v_1, \ldots, v_k \rangle \)

  \[
  = \sum_{i=1}^{k} w(v_{i-1}, v_i)
  = \text{sum of edge weights on path } p.
  \]

- **Shortest path weight:**

  \[
  \delta(u, v) = \begin{cases} 
  \min \left\{ w(p) : u \xrightarrow{p} v \right\} & \text{if there exists a path } u \xrightarrow{p} v, \\
  \infty & \text{otherwise}.
  \end{cases}
  \]
Various Problems

- **Single-source shortest-paths problem**
  - Given source node $s$ to all nodes from $V$

- **Single-destination shortest-paths problem**
  - From all nodes in $V$ to a destination $u$

- **Single-pair shortest-path problem**
  - Shortest path between $u$ and $v$

- **All-pairs shortest-paths problem**
  - Shortest paths between all pairs of nodes
Example
Negative-weight Edges

- Some edges may have negative weights

- If there is a negative cycle reachable from $s$:
  - Shortest path is no longer well-defined
  - Example

- Otherwise, it is fine
Cycles

- Shortest path cannot have cycles inside
  - Negative cycles: already eliminated
  - Positive cycles: can be removed
  - 0-weight cycles: can be removed
- A path with no cycles inside is also called a *simple path*.

No-Cycle Theorem:
Given any two vertices of graph, there exists a shortest path between them with no cycle.
Optimal Substructure Property

Optimal Substructure Property (Theorem): If \((u_1, u_2, \ldots, u_m)\) is a shortest path from \(u_1\) to \(u_m\), then any sub-path \((u_i, \ldots, u_j)\) is also a shortest path.

- **Proof**
  - Proof by contradiction.
  - If there is a shorter path \(\pi\) from \(u_i\) to \(u_j\), then the path \((u_1, \ldots, u_i) \circ \pi \circ (u_j, \ldots, u_m)\) is a shorter path from \(u_1\) to \(u_m\). Contradiction.
Shortest-paths Tree

- For every node \( v \in V \), \( \pi[v] \) is the predecessor of \( v \) in a shortest path from source \( s \) to \( v \)
  - Nil if does not exist

- A shortest-path tree
  - Root is source \( s \)
  - Edges are \( (\pi[v], v) \)

- The shortest path between \( s \) and \( v \) is the unique tree path from root \( s \) to \( v \).
Example: directed graph
Example: undirected graph

- Key property:
  - Given shortest path tree rooted at s (\(v_1\) in this example), one can obtain the shortest path from s to every other vertex connected to it.
Shortest Path Tree

Question:

- Given a shortest path tree of G rooted at s, how fast can we report the shortest path distance from s to all other vertices in G?

\[ O(V) \]

Key property:

- Given shortest path tree rooted at s (\(v_1\) in this example), one can obtain the shortest path from s to every other vertex connected to it.

The output of shortest-path algorithms usually contains both shortest-path tree and distances.
Shortest Path Tree

Related to minimum spanning tree?

Shortest Path Tree:

Minimum Spanning Tree:
Goal:

- **Input:**
  - a weighted graph $G = (V, E)$, with *positive weights*!
  - and a source node $v_s \in V$

- **Output:**
  - For every vertex $v \in V$, 
    - $v.\text{distance} = \delta (v_s, v)$
    - $v.\text{parent} = \pi[v]$
  - Shortest-paths tree induced by $v.\text{parent}$
Breadth-First Search

- Recall BFS:
  - Shortest path for unweighted graph (i.e., each edge has weight 1).

- Would the same idea work for weighted graph?
We can no longer guarantee that at the time the algorithm first discovers a node, the distance is shortest.

BFS is only for shortest number of edges to reach a node!

How to guarantee that when we first reach a node, the distance is shortest?
Dijkstra Algorithm

procedure DijkstraShortestPath(G, v_s)
1    U ← V(G) − {v_s}; /* V(G) = set of vertices of graph G */
2    v_s.parent ← NULL;
3    v_s.distance ← 0;
4    while (U ≠ ∅) and (∃ edge from (V(G) − U) to U) do
5       (v_i, v_j) ← edge from V(G) − U to U which minimizes
6               v_i.distance + weight(v_i, v_j);
7       v_j.parent ← v_i;
8       v_j.distance ← v_i.distance + weight(v_i, v_j);
9    end
Intuition:

- If $v_i \cdot distance = \delta(v_s,v_i)$
- Then $v_i \cdot distance + weight(v_i,v_j)$ is the shortest distance to reach $v_j$ through $v_i$.
- Note, this does not have to be the same as shortest distance to $v_j$. 
Example
Correctness

- **Invariance:**
  - At the beginning of the While-loop, all vertices already discovered have correct shortest distance value.

- **Prove that this invariance is maintained:**
  - Base case: in the first iteration, only $v_s$ is discovered, and this invariance holds.
  
  - Inductive step: If the invariance holds at the beginning of the $k$-th iteration, then it holds at the end of $k$-th iteration (i.e., it holds at the beginning of $(k + 1)$-th iteration).
Proof of Induction Step

- Let \((v_i, v_j)\) as identified in Line 5 of the algorithm.
  - Let \(P\) be the shortest path from \(v_s\) to \(v_i\).
  - \(v_i.\ distance + weight(v_s, v_i)\) is the weight of the path \(P \cup \{(v_i, v_j)\}\).
- Proof that any other path from \(v_s\) to \(v_j\) has larger weight.
  - Consider any other path \(P'\) from \(v_s\) to \(v_j\).
  - Let \((x, y)\) be the first edge in \(P'\) that connects a vertex from \(V - U\) to a vertex in \(U\).
  - Argue that the weight of subpath from \(v_s\) to \(y\) is at least \(v_i.\ distance + weight(v_s, v_i)\).
  - Hence the weight of \(P'\) is larger than that of \(P \cup \{(v_i, v_j)\}\).
- Done.
Why do we require that the weights are all positive?

Example.
procedure DijkstraShortestPath(G, v_s)
1 \( U \leftarrow V(G) - \{v_s\} \); /* \( V(G) = \) set of vertices of graph \( G \) */
2 \( v_s.parent \leftarrow \text{NULL}; \)
3 \( v_s.distance \leftarrow 0; \)
4 while \( (U \neq \emptyset) \) and \( (\exists \) edge from \( (V(G) - U) \) to \( U \) \) do
5 \( (v_i, v_j) \leftarrow \) edge from \( V(G) - U \) to \( U \) which minimizes \( v_i.distance + \text{weight}(v_i, v_j); \)
6 \( v_j.parent \leftarrow v_i; \)
7 \( v_j.distance \leftarrow v_i.distance + \text{weight}(v_i, v_j); \)
8 \( U \leftarrow U - \{v_j\}; \)
9 end
Running Time Analysis

- Naïve implementation:
  - Spend $O(E)$ time to identify $(v_i, v_j)$ in Line 5.
  - Total time: $O(VE)$

- How to identify $(v_i, v_j)$ more efficiently?
  - First improvement: Storing cost at vertices.
    - $v$.distance:
      - current estimate of shortest path weight from $v_s$ to $v$. 

CSE 2331 / 5331
Storing Cost at Vertices

```plaintext
procedure DijkstraShortestPath(G, v_s)
    1 $U \leftarrow V(G)$; /* $V(G) =$ set of vertices of graph $G$ */
    2 foreach $v_i \in V(G) - \{v_s\}$ do $v_i$.distance $\leftarrow \infty$;
    3 $v_s$.distance $\leftarrow 0$;
    4 $v_s$.parent $\leftarrow$ NULL;
    5 while ($U \neq \emptyset$) and ($v_i$.distance $< \infty$ for some $v_i \in U$) do
        6 $v_j \leftarrow v_i \in U$ with minimum $v_i$.distance;
        7 $U \leftarrow U - \{v_j\}$; /* Remove $v_j$ from $U$ */
        8 /* ($v_j$, $v_j$.parent) is a shortest path edge */
        9 foreach edge ($v_j$, $v_k$) incident on $v_j$ do
            10 newDist $\leftarrow v_j$.distance + weight($v_j$, $v_k$);
            11 if ($v_k \in U$ and newDist $< v_k$.distance) then
                12 $v_k$.parent $\leftarrow v_j$;
                13 $v_k$.distance $\leftarrow$ newDist;
            14 end
        15 end
end
```
Running Time Analysis

- First improvement:
  - Spend $O(V)$ time to identify $(v_i, v_j)$ in Line 5.
  - Total time: $O(V^2)$
  - Note: the weight stored at $v_j$ is NOT the smallest weight of edge connecting $v_j$ to any visited vertex
    - That is how Prim’s MST algorithm works.

- How to identify $(v_i, v_j)$ more efficiently?
  - Second improvement: Use Priority Queue to store / maintain vertex costs!
    - $v$.distance: current estimate of shortest path weight from $v_s$ to $v$. 
procedure DijkstraShortestPath(G, v_s)
1 foreach v_i ∈ V(G) − {v_s} do Q.Insert(v_i, ∞);
2 Q.Insert(v_s, 0); /* Q is a priority queue of vertices */
3 v_s.parent ← NULL;
4 v_s.distance ← 0;
5 while Q.IsNotEmpty() and (Q.MinKey() ≠ ∞) do
6     v_j ← Q.DeleteMin();
7     /* (v_j, v_j.parent) is a shortest path edge */
8     foreach edge (v_j, v_k) incident on v_j do
9         newDist ← v_j.distance + weight(v_j, v_k);
10        if (v_k is in U and newDist < v_k.distance) then
11            v_k.parent ← v_j;
12            Q.DecreaseKey(v_k, newDist);
13            v_k.distance ← newDist;
14     end
15 end
Running Time Analysis

- **Priority Queue: \( Q \):**
  - Total size: \( O(V) \)
  - \# \( Q.\text{insert} \): \( O(V) \)
    - Total time: \( O(V \ lg V) \)
  - \# \( Q.\text{DeleteMin} \): \( O(V) \)
    - Total time: \( O(V \ lg V) \)
  - \# \( Q.\text{isEmpty}() \) and \# \( Q.\text{MinKey}() \): \( O(V) \)
    - Total time: \( O(V) \)
  - \# \( Q.\text{DecreaseKey} \): \( O(E) \)
    - Total time: \( O(E \ lg V) \)

Total time: \( O((V + E) \ lg V) \)
Remarks

- Similar idea as breadth first search:
  - Greedy type of algorithm
  - Guarantees that when we first discover a node, the distance is the correct shortest path weight
- Similar to Prim’s Alg for MST:
  - But the cost at each vertex is defined differently.
- Also works for directed graphs
- But require weights to be positive
- $O(V \log V + E \log V) = O((V + E) \log V)$
  - Can be improved to $O(E + V \log V)$ by using a better implementation of priority queue (Fibonacci heap)