算法设计与分析(2024年春季学期)
Graph Algorithms

授课老师: 栗师
南京大学计算机科学与技术系
Outline

1. Minimum Spanning Tree
   - Kruskal’s Algorithm
   - Reverse-Kruskal’s Algorithm
   - Prim’s Algorithm

2. Single Source Shortest Paths
   - Dijkstra’s Algorithm

3. Shortest Paths in Graphs with Negative Weights

4. All-Pair Shortest Paths and Floyd-Warshall

5. Minimum Cost Arborescence
Def. Given a connected graph $G = (V, E)$, a spanning tree $T = (V, F)$ of $G$ is a sub-graph of $G$ that is a tree including all vertices $V$. 
Lemma  Let $T = (V, F)$ be a subgraph of $G = (V, E)$. The following statements are equivalent:

- $T$ is a spanning tree of $G$;
- $T$ is acyclic and connected;
- $T$ is connected and has $n - 1$ edges;
- $T$ is acyclic and has $n - 1$ edges;
- $T$ is minimally connected: removal of any edge disconnects it;
- $T$ is maximally acyclic: addition of any edge creates a cycle;
- $T$ has a unique simple path between every pair of nodes.
Minimum Spanning Tree (MST) Problem

**Input:** Graph $G = (V, E)$ and edge weights $w : E \rightarrow \mathbb{R}$

**Output:** the spanning tree $T$ of $G$ with the minimum total weight
Recall: Steps of Designing A Greedy Algorithm

- Design a “reasonable” strategy
- Prove that the reasonable strategy is “safe” (key, usually done by “exchanging argument”)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually trivial)

Def. A choice is “safe” if there is an optimum solution that is “consistent” with the choice

Two Classic Greedy Algorithms for MST

- Kruskal’s Algorithm
- Prim’s Algorithm
Outline

1. Minimum Spanning Tree
   - Kruskal’s Algorithm
   - Reverse-Kruskal’s Algorithm
   - Prim’s Algorithm

2. Single Source Shortest Paths
   - Dijkstra’s Algorithm

3. Shortest Paths in Graphs with Negative Weights

4. All-Pair Shortest Paths and Floyd-Warshall

5. Minimum Cost Arborescence
Q: Which edge can be safely included in the MST?

A: The edge with the smallest weight (lightest edge).
**Lemma**  It is safe to include the lightest edge: there is a minimum spanning tree, that contains the lightest edge.

**Proof.**
- Take a minimum spanning tree $T$
- Assume the lightest edge $e^*$ is not in $T$
- There is a unique path in $T$ connecting $u$ and $v$
- Remove any edge $e$ in the path to obtain tree $T'$
- $w(e^*) \leq w(e) \implies w(T') \leq w(T)$: $T'$ is also a MST
Is the Residual Problem Still a MST Problem?

- Residual problem: find the minimum spanning tree that contains edge \((g, h)\)
- Contract the edge \((g, h)\)
- Residual problem: find the minimum spanning tree in the contracted graph
Contraction of an Edge \((u, v)\)

- Remove \(u\) and \(v\) from the graph, and add a new vertex \(u^*\)
- Remove all edges \((u, v)\) from \(E\)
- For every edge \((u, w)\) \(\in E\), \(w \neq v\), change it to \((u^*, w)\)
- For every edge \((v, w)\) \(\in E\), \(w \neq u\), change it to \((u^*, w)\)
- May create parallel edges! E.g. : two edges \((i, g^*)\)
Greedy Algorithm

Repeat the following step until $G$ contains only one vertex:

1. Choose the lightest edge $e^*$, add $e^*$ to the spanning tree
2. Contract $e^*$ and update $G$ be the contracted graph

Q: What edges are removed due to contractions?

A: Edge $(u, v)$ is removed if and only if there is a path connecting $u$ and $v$ formed by edges we selected
Greedy Algorithm

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

1. \(F \leftarrow \emptyset\)
2. sort edges in \(E\) in non-decreasing order of weights \(w\)
3. **for** each edge \((u, v)\) in the order **do**
4. **if** \(u\) and \(v\) are not connected by a path of edges in \(F\) **then**
5. \(F \leftarrow F \cup \{(u, v)\}\)
6. **return** \((V, F)\)
Kruskal’s Algorithm: Example

Sets: \{a, b, c, i, f, g, h, d, e\}
Kruskal’s Algorithm: Efficient Implementation of Greedy Algorithm

MST-Kruskal($G, w$)

1: $F \leftarrow \emptyset$
2: $S \leftarrow \{\{v\} : v \in V\}$
3: sort the edges of $E$ in non-decreasing order of weights $w$
4: for each edge $(u, v) \in E$ in the order do
5: $S_u \leftarrow$ the set in $S$ containing $u$
6: $S_v \leftarrow$ the set in $S$ containing $v$
7: if $S_u \neq S_v$ then
8: $F \leftarrow F \cup \{(u, v)\}$
9: $S \leftarrow S \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$
10: return $(V, F)$
Running Time of Kruskal’s Algorithm

MST-Kruskal($G, w$)

1: $F \leftarrow \emptyset$
2: $S \leftarrow \{\{v\} : v \in V\}$
3: sort the edges of $E$ in non-decreasing order of weights $w$
4: for each edge $(u, v) \in E$ in the order do
5: $S_u \leftarrow$ the set in $S$ containing $u$
6: $S_v \leftarrow$ the set in $S$ containing $v$
7: if $S_u \neq S_v$ then
8: $F \leftarrow F \cup \{(u, v)\}$
9: $S \leftarrow S \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$
10: return $(V, F)$

Use union-find data structure to support 2, 5, 6, 7, 9.
Union-Find Data Structure

- $V$: ground set
- We need to maintain a partition of $V$ and support following operations:
  - Check if $u$ and $v$ are in the same set of the partition
  - Merge two sets in partition
• $V = \{1, 2, 3, \cdots, 16\}$

• Partition: $\{2, 3, 5, 9, 10, 12, 15\}$, $\{1, 7, 13, 16\}$, $\{4, 8, 11\}$, $\{6, 14\}$

• $par[i]$: parent of $i$, ($par[i] = \bot$ if $i$ is a root).
Q: how can we check if $u$ and $v$ are in the same set?
A: Check if $\text{root}(u) = \text{root}(v)$.

$\text{root}(u)$: the root of the tree containing $u$

Merge the trees with root $r$ and $r'$: $\text{par}[r] \leftarrow r'$. 
Union-Find Data Structure

root(\(v\))

1: \textbf{if} \(par[v] = \bot\) \textbf{then}
2: \quad \textbf{return} \(v\)
3: \textbf{else}
4: \quad \textbf{return} root(par[v])

--

text

Problem: the tree might too deep; running time might be large

Improvement: all vertices in the path directly point to the root, saving time in the future.
root($v$)

1: if $par[v] = \bot$ then
2: return $v$
3: else
4: $par[v] \leftarrow \text{root}(par[v])$
5: return $par[v]$

![Diagram of Union-Find Data Structure](image)
MST-Kruskal($G$, $w$)

1. $F \leftarrow \emptyset$
2. $S \leftarrow \{ \{v\} : v \in V \}$
3. sort the edges of $E$ in non-decreasing order of weights $w$
4. for each edge $(u, v) \in E$ in the order do
5.   $S_u \leftarrow$ the set in $S$ containing $u$
6.   $S_v \leftarrow$ the set in $S$ containing $v$
7.   if $S_u \neq S_v$ then
8.      $F \leftarrow F \cup \{ (u, v) \}$
9.      $S \leftarrow S \setminus \{ S_u \} \setminus \{ S_v \} \cup \{ S_u \cup S_v \}$
10. return $(V, F)$
MST-Kruskal($G$, $w$)

1: $F \leftarrow \emptyset$

2: for every $v \in V$ do: $\text{par}[v] \leftarrow \bot$

3: sort the edges of $E$ in non-decreasing order of weights $w$

4: for each edge $(u, v) \in E$ in the order do

5: $u' \leftarrow \text{root}(u)$

6: $v' \leftarrow \text{root}(v)$

7: if $u' \neq v'$ then

8: $F \leftarrow F \cup \{(u, v)\}$

9: $\text{par}[u'] \leftarrow v'$

10: return $(V, F)$

- 2, 5, 6, 7, 9 takes time $O(m\alpha(n))$
- $\alpha(n)$ is very slow-growing: $\alpha(n) \leq 4$ for $n \leq 10^{80}$.
- Running time = time for 3 = $O(m \lg n)$. 
Assumption  Assume all edge weights are different.

Lemma  An edge $e \in E$ is not in the MST, if and only if there is cycle $C$ in $G$ in which $e$ is the heaviest edge.

- $(i, g)$ is not in the MST because of cycle $(i, c, f, g)$
- $(e, f)$ is in the MST because no such cycle exists
Outline

1. **Minimum Spanning Tree**
   - Kruskal’s Algorithm
   - Reverse-Kruskal’s Algorithm
   - Prim’s Algorithm

2. **Single Source Shortest Paths**
   - Dijkstra’s Algorithm

3. **Shortest Paths in Graphs with Negative Weights**

4. **All-Pair Shortest Paths and Floyd-Warshall**

5. **Minimum Cost Arborescence**
Two Methods to Build a MST

1. Start from $F \leftarrow \emptyset$, and add edges to $F$ one by one until we obtain a spanning tree.

2. Start from $F \leftarrow E$, and remove edges from $F$ one by one until we obtain a spanning tree.

Q: Which edge can be safely excluded from the MST?

A: The heaviest non-bridge edge.

Def. A bridge is an edge whose removal disconnects the graph.
Lemma  It is safe to exclude the heaviest non-bridge edge: there is a MST that does not contain the heaviest non-bridge edge.
Reverse Kruskal’s Algorithm

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

1. \(F \leftarrow E\)
2. sort \(E\) in non-increasing order of weights
3. for every \(e\) in this order do
4. if \((V, F \setminus \{e\})\) is connected then
5. \(F \leftarrow F \setminus \{e\}\)
6. return \((V, F)\)
Reverse Kruskal’s Algorithm: Example
Outline

1. **Minimum Spanning Tree**
   - Kruskal’s Algorithm
   - Reverse-Kruskal’s Algorithm
   - Prim’s Algorithm

2. **Single Source Shortest Paths**
   - Dijkstra’s Algorithm

3. **Shortest Paths in Graphs with Negative Weights**

4. **All-Pair Shortest Paths and Floyd-Warshall**

5. **Minimum Cost Arborescence**
Recall the greedy strategy for Kruskal’s algorithm: choose the edge with the smallest weight.

Greedy strategy for Prim’s algorithm: choose the lightest edge incident to $a$. 
Lemma  It is safe to include the lightest edge incident to \( a \).

Proof.

- Let \( T \) be a MST
- Consider all components obtained by removing \( a \) from \( T \)
- Let \( e^* \) be the lightest edge incident to \( a \) and \( e^* \) connects \( a \) to component \( C \)
- Let \( e \) be the edge in \( T \) connecting \( a \) to \( C \)
- \( T' = T \setminus \{e\} \cup \{e^*\} \) is a spanning tree with \( w(T') \leq w(T) \)
Prim’s Algorithm: Example
Greedy Algorithm

**MST-Greedy1**($G, w$)

1: $S \leftarrow \{s\}$, where $s$ is arbitrary vertex in $V$
2: $F \leftarrow \emptyset$
3: **while** $S \neq V$ **do**
4: $(u, v) \leftarrow$ lightest edge between $S$ and $V \setminus S$, where $u \in S$ and $v \in V \setminus S$
5: $S \leftarrow S \cup \{v\}$
6: $F \leftarrow F \cup \{(u, v)\}$
7: **return** $(V, F)$

- **Running time of naive implementation:** $O(nm)$
Prim’s Algorithm: Efficient Implementation of Greedy Algorithm

For every $v \in V \setminus S$ maintain

- $d[v] = \min_{u \in S: (u,v) \in E} w(u,v)$:
  the weight of the lightest edge between $v$ and $S$

- $\pi[v] = \arg \min_{u \in S: (u,v) \in E} w(u,v)$:
  $(\pi[v], v)$ is the lightest edge between $v$ and $S$
Prim’s Algorithm: Efficient Implementation of Greedy Algorithm

For every $v \in V \setminus S$ maintain

- $d[v] = \min_{u \in S: (u,v) \in E} w(u, v)$: the weight of the lightest edge between $v$ and $S$
- $\pi[v] = \arg \min_{u \in S: (u,v) \in E} w(u, v)$: $(\pi[v], v)$ is the lightest edge between $v$ and $S$

In every iteration

- Pick $u \in V \setminus S$ with the smallest $d[u]$ value
- Add $(\pi[u], u)$ to $F$
- Add $u$ to $S$, update $d$ and $\pi$ values.
Prim’s Algorithm

### MST-Prim($G, w$)

1: $s \leftarrow$ arbitrary vertex in $G$
2: $S \leftarrow \emptyset$, $d(s) \leftarrow 0$ and $d[v] \leftarrow \infty$ for every $v \in V \setminus \{s\}$
3: **while** $S \neq V$ **do**
4: $u \leftarrow$ vertex in $V \setminus S$ with the minimum $d[u]$
5: $S \leftarrow S \cup \{u\}$
6: **for** each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
7: **if** $w(u, v) < d[v]$ **then**
8: $d[v] \leftarrow w(u, v)$
9: $\pi[v] \leftarrow u$
10: **return** $\{(u, \pi[u])|u \in V \setminus \{s\}\}$
Example
**Prim’s Algorithm**

For every $v \in V \setminus S$ maintain

- $d[v] = \min_{u \in S : (u,v) \in E} w(u,v)$: the weight of the lightest edge between $v$ and $S$

- $\pi[v] = \arg\min_{u \in S : (u,v) \in E} w(u,v)$: $(\pi[v], v)$ is the lightest edge between $v$ and $S$

In every iteration

- Pick $u \in V \setminus S$ with the smallest $d[u]$ value
  - `extract_min`

- Add $(\pi[u], u)$ to $F$

- Add $u$ to $S$, update $d$ and $\pi$ values.
  - `decrease_key`

Use a priority queue to support the operations
Def. A **priority queue** is an abstract data structure that maintains a set $U$ of elements, each with an associated key value, and supports the following operations:

- **insert**($v, key_value$): insert an element $v$, whose associated key value is $key_value$.
- **decrease_key**($v, new_key_value$): decrease the key value of an element $v$ in queue to $new_key_value$
- **extract_min**(): return and remove the element in queue with the smallest key value
Prim’s Algorithm

\textbf{MST-Prim}(G, w)

1: \( s \leftarrow \) arbitrary vertex in \( G \)
2: \( S \leftarrow \emptyset, d(s) \leftarrow 0 \) and \( d[v] \leftarrow \infty \) for every \( v \in V \setminus \{s\} \)
3: 
4: \textbf{while} \( S \neq V \) \textbf{do}
5: \( u \leftarrow \) vertex in \( V \setminus S \) with the minimum \( d[u] \)
6: \( S \leftarrow S \cup \{u\} \)
7: \textbf{for} each \( v \in V \setminus S \) such that \( (u, v) \in E \) \textbf{do}
8: \hspace{1em} \textbf{if} \( w(u, v) < d[v] \) \textbf{then}
9: \hspace{2em} \( d[v] \leftarrow w(u, v) \)
10: \hspace{2em} \( \pi[v] \leftarrow u \)
11: \textbf{return} \( \{(u, \pi[u])|u \in V \setminus \{s\}\} \)
Prim’s Algorithm Using Priority Queue

MST-Prim($G, w$)

1: $s \leftarrow$ arbitrary vertex in $G$
2: $S \leftarrow \emptyset$, $d(s) \leftarrow 0$ and $d[v] \leftarrow \infty$ for every $v \in V \setminus \{s\}$
3: $Q \leftarrow$ empty queue, for each $v \in V$: $Q.insert(v, d[v])$
4: while $S \neq V$ do
5: \hspace{1em} $u \leftarrow Q.extract\_min()$
6: \hspace{1em} $S \leftarrow S \cup \{u\}$
7: \hspace{1em} for each $v \in V \setminus S$ such that $(u, v) \in E$ do
8: \hspace{2em} if $w(u, v) < d[v]$ then
9: \hspace{3em} $d[v] \leftarrow w(u, v)$, $Q.decrease\_key(v, d[v])$
10: \hspace{3em} $\pi[v] \leftarrow u$
11: return $\{(u, \pi[u]) | u \in V \setminus \{s\}\}$
Running Time of Prim’s Algorithm Using Priority Queue

\[ O(n) \times (\text{time for extract\_min}) + O(m) \times (\text{time for decrease\_key}) \]

<table>
<thead>
<tr>
<th>concrete DS</th>
<th>extract_min</th>
<th>decrease_key</th>
<th>overall time</th>
</tr>
</thead>
<tbody>
<tr>
<td>heap</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(m \log n) )</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
<td>( O(n \log n + m) )</td>
</tr>
</tbody>
</table>
**Assumption**  Assume all edge weights are different.

**Lemma**  \((u, v)\) is in MST, if and only if there exists a cut \((U, V \setminus U)\), such that \((u, v)\) is the lightest edge between \(U\) and \(V \setminus U\).

- \((c, f)\) is in MST because of cut \(\{a, b, c, i\}, V \setminus \{a, b, c, i\}\)
- \((i, g)\) is not in MST because no such cut exists
“Evidence” for \( e \in \text{MST} \) or \( e \notin \text{MST} \)

**Assumption** Assume all edge weights are different.

- \( e \in \text{MST} \iff \text{there is a cut in which } e \text{ is the lightest edge} \)
- \( e \notin \text{MST} \iff \text{there is a cycle in which } e \text{ is the heaviest edge} \)

Exactly one of the following is true:

- There is a cut in which \( e \) is the lightest edge
- There is a cycle in which \( e \) is the heaviest edge

Thus, the minimum spanning tree is unique with assumption.
Outline

1. Minimum Spanning Tree
   - Kruskal’s Algorithm
   - Reverse-Kruskal’s Algorithm
   - Prim’s Algorithm

2. Single Source Shortest Paths
   - Dijkstra’s Algorithm

3. Shortest Paths in Graphs with Negative Weights

4. All-Pair Shortest Paths and Floyd-Warshall

5. Minimum Cost Arborescence
<table>
<thead>
<tr>
<th>algorithm</th>
<th>graph</th>
<th>weights</th>
<th>SS?</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple DP</td>
<td>DAG</td>
<td>R</td>
<td>SS</td>
<td>(O(n + m))</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>U/D</td>
<td>(\mathbb{R}_{\geq 0})</td>
<td>SS</td>
<td>(O(n \log n + m))</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>U/D</td>
<td>R</td>
<td>SS</td>
<td>(O(nm))</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>U/D</td>
<td>R</td>
<td>AP</td>
<td>(O(n^3))</td>
</tr>
</tbody>
</table>

- DAG = directed acyclic graph
- U = undirected
- D = directed
- SS = single source
- AP = all pairs
**s-t Shortest Paths**

**Input:** (directed or undirected) graph $G = (V, E)$, $s, t \in V$

$w : E \rightarrow \mathbb{R}_{\geq 0}$

**Output:** shortest path from $s$ to $t$
Single Source Shortest Paths

**Input:** directed graph $G = (V, E)$, $s \in V$

$w : E \rightarrow \mathbb{R}_{\geq 0}$

**Output:** shortest paths from $s$ to all other vertices $v \in V$

Reason for Considering Single Source Shortest Paths

- We do not know how to solve $s$-$t$ shortest path problem more efficiently than solving single source shortest path problem

- Shortest paths in directed graphs is more general than in undirected graphs: we can replace every undirected edge with two anti-parallel edges of the same weight
Single Source Shortest Paths

**Input:** directed graph $G = (V, E)$, $s \in V$

$w : E \rightarrow \mathbb{R}_{\geq 0}$

**Output:** $\pi[v], v \in V \setminus s$: the parent of $v$ in shortest path tree

$d[v], v \in V \setminus s$: the length of shortest path from $s$ to $v$
Q: How to compute shortest paths from $s$ when all edges have weight 1?

A: Breadth first search (BFS) from source $s$
Assumption Weights $w(u, v)$ are integers (w.l.o.g).

- An edge of weight $w(u, v)$ is equivalent to a path of $w(u, v)$ unit-weight edges

Shortest Path Algorithm by Running BFS

1. replace $(u, v)$ of length $w(u, v)$ with a path of $w(u, v)$ unit-weight edges, for every $(u, v) \in E$
2. run BFS virtually
3. $\pi[v] \leftarrow$ vertex from which $v$ is visited
4. $d[v] \leftarrow$ index of the level containing $v$

- Problem: $w(u, v)$ may be too large!
Shortest Path Algorithm by Running BFS Virtually

1: $S \leftarrow \{s\}, d(s) \leftarrow 0$
2: while $|S| \leq n$ do
3: find a $v \notin S$ that minimizes $\min_{u \in S: (u, v) \in E} \{d[u] + w(u, v)\}$
4: $S \leftarrow S \cup \{v\}$
5: $d[v] \leftarrow \min_{u \in S: (u, v) \in E} \{d[u] + w(u, v)\}$
Virtual BFS: Example

Time 10
Outline

1. Minimum Spanning Tree
   - Kruskal’s Algorithm
   - Reverse-Kruskal’s Algorithm
   - Prim’s Algorithm

2. Single Source Shortest Paths
   - Dijkstra’s Algorithm

3. Shortest Paths in Graphs with Negative Weights

4. All-Pair Shortest Paths and Floyd-Warshall

5. Minimum Cost Arborescence
Dijkstra’s Algorithm

Dijkstra\( (G, w, s) \)

1: \( S \leftarrow \emptyset \), \( d(s) \leftarrow 0 \) and \( d[v] \leftarrow \infty \) for every \( v \in V \setminus \{s\} \)
2: while \( S \neq V \) do
3: \( u \leftarrow \) vertex in \( V \setminus S \) with the minimum \( d[u] \)
4: add \( u \) to \( S \)
5: for each \( v \in V \setminus S \) such that \( (u, v) \in E \) do
6: if \( d[u] + w(u, v) < d[v] \) then
7: \( d[v] \leftarrow d[u] + w(u, v) \)
8: \( \pi[v] \leftarrow u \)
9: return \( (d, \pi) \)

- Running time = \( O(n^2) \)
Improved Running Time using Priority Queue

\textbf{Dijkstra}(G, w, s)

1: \(s \leftarrow \) arbitrary vertex in \(G\)
2: \(S \leftarrow \emptyset, d(s) \leftarrow 0 \) and \(d[v] \leftarrow \infty\) for every \(v \in V \setminus \{s\}\)
3: \(Q \leftarrow\) empty queue, for each \(v \in V:\ Q.\text{insert}(v, d[v])\)
4: \textbf{while} \(S \neq V\) \textbf{do}
5: \(u \leftarrow Q.\text{extract\_min}()\)
6: \(S \leftarrow S \cup \{u\}\)
7: \textbf{for} each \(v \in V \setminus S\) such that \((u, v) \in E\) \textbf{do}
8: \textbf{if} \(d[u] + w(u, v) < d[v]\) \textbf{then}
9: \(d[v] \leftarrow d[u] + w(u, v),\ Q.\text{decrease\_key}(v, d[v])\)
10: \(\pi[v] \leftarrow u\)
11: \textbf{return} \((\pi, d)\)
Recall: Prim’s Algorithm for MST

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

1: \(s \leftarrow\) arbitrary vertex in \(G\)
2: \(S \leftarrow \emptyset, d(s) \leftarrow 0\) and \(d[v] \leftarrow \infty\) for every \(v \in V \setminus \{s\}\)
3: \(Q \leftarrow\) empty queue, for each \(v \in V\): \(Q.insert(v, d[v])\)
4: **while** \(S \neq V\) **do**
5: \(u \leftarrow Q.extract\_min()\)
6: \(S \leftarrow S \cup \{u\}\)
7: **for** each \(v \in V \setminus S\) such that \((u, v) \in E\) **do**
8: \(\text{if } w(u, v) < d[v] \text{ then}\)
9: \(d[v] \leftarrow w(u, v), Q.decrease\_key(v, d[v])\)
10: \(\pi[v] \leftarrow u\)
11: **return** \(\{(u, \pi[u])|u \in V \setminus \{s\}\}\)
Improved Running Time

Running time:
\[ O(n) \times (\text{time for extract\_min}) + O(m) \times (\text{time for decrease\_key}) \]

<table>
<thead>
<tr>
<th>Priority-Queue</th>
<th>extract_min</th>
<th>decrease_key</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(m \log n) )</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
<td>( O(n \log n + m) )</td>
</tr>
</tbody>
</table>
Outline

1. Minimum Spanning Tree
   - Kruskal’s Algorithm
   - Reverse-Kruskal’s Algorithm
   - Prim’s Algorithm

2. Single Source Shortest Paths
   - Dijkstra’s Algorithm

3. Shortest Paths in Graphs with Negative Weights

4. All-Pair Shortest Paths and Floyd-Warshall

5. Minimum Cost Arborescence
**Input:** directed graph \( G = (V, E) \), \( s \in V \)
assumee all vertices are reachable from \( s \)
\( w : E \rightarrow \mathbb{R} \)

**Output:** shortest paths from \( s \) to all other vertices \( v \in V \)

- In transition graphs, negative weights make sense
- If we sell a item: ‘having the item’ \( \rightarrow \) ‘not having the item’, weight is negative (we gain money)
- Dijkstra’s algorithm does not work any more!
Dijkstra’s Algorithm Fails if We Have Negative Weights
Q: What is the length of the shortest path from \( s \) to \( d \)?

A: \(-\infty\)

Def. A negative cycle is a cycle in which the total weight of edges is negative.

Q: What is the length of the shortest simple path from \( s \) to \( d \)?

A: 1
Unfortunately, computing the shortest simple path between two vertices is an \textbf{NP-hard} problem.

\textbf{Dealing with Negative Cycles}

- We need to compute the shortest paths, among both simple and complex paths.
- Hardest: output $-\infty$ as a distance
- Easier: if negative cycle exists, allow algorithm to report “negative cycle exists” without computing distances
- Easiest: assume negative cycles do not exist; all shortest paths are automatically simple paths
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Graph Form</th>
<th>Weights</th>
<th>SS?</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple DP</td>
<td>DAG</td>
<td>( \mathbb{R} )</td>
<td>SS</td>
<td>( O(n + m) )</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>U/D</td>
<td>( \mathbb{R}_{\geq 0} )</td>
<td>SS</td>
<td>( O(n \log n + m) )</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>U/D</td>
<td>( \mathbb{R} )</td>
<td>SS</td>
<td>( O(nm) )</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>U/D</td>
<td>( \mathbb{R} )</td>
<td>AP</td>
<td>( O(n^3) )</td>
</tr>
</tbody>
</table>

- DAG = directed acyclic graph
- U = undirected
- D = directed
- SS = single source
- AP = all pairs
Single Source Shortest Paths, Weights May be Negative

**Input:** directed graph $G = (V, E)$, $s \in V$

- assume all vertices are reachable from $s$
- $w : E \rightarrow \mathbb{R}$

**Output:** shortest paths from $s$ to all other vertices $v \in V$

- first try: $f[v]$: length of shortest path from $s$ to $v$
- issue: do not know in which order we compute $f[v]$’s
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots , n - 1\}$, $v \in V$: length of shortest path from $s$ to $v$ that uses at most $\ell$ edges
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n - 1\}$, $v \in V$: length of shortest path from $s$ to $v$ that uses at most $\ell$ edges
- $f^2[a] = 6$
- $f^3[a] = 2$

$$f^\ell[v] = \begin{cases} 
0 & \text{if } \ell = 0, v = s \\
\infty & \text{if } \ell = 0, v \neq s \\
\min \left\{ \min_{u : (u,v) \in E} \left( f^{\ell-1}[u] + w(u,v) \right) \right\} & \text{if } \ell > 0
\end{cases}$$
Dynamic Programming: Example

\[ f^0 \]

\[ f^1 \]

\[ f^2 \]

\[ f^3 \]

\[ f^4 \]

\[ \text{length-0 edge} \]
**dynamic-programming**(*G, w, s*)

1. \( f^0[s] \leftarrow 0 \) and \( f^0[v] \leftarrow \infty \) for any \( v \in V \setminus \{s\} \)
2. **for** \( \ell \leftarrow 1 \) to \( n - 1 \) **do**
3. copy \( f^{\ell-1} \rightarrow f^\ell \)
4. **for** each \((u, v) \in E\) **do**
5. \[ \text{if } f^{\ell-1}[u] + w(u, v) < f^\ell[v] \text{ then} \]
6. \[ f^\ell[v] \leftarrow f^{\ell-1}[u] + w(u, v) \]
7. **return** \((f^{n-1}[v])_{v \in V}\)

**Obs.** Assuming there are no negative cycles, then a shortest path contains at most \( n - 1 \) edges

**Proof.**

If there is a path containing at least \( n \) edges, then it contains a cycle. Removing the cycle gives a path with the same or smaller length. \( \square \)
Bellman-Ford Algorithm

Bellman-Ford\((G, w, s)\)

1: \(f[s] \leftarrow 0\) and \(f[v] \leftarrow \infty\) for any \(v \in V \setminus \{s\}\)
2: for \(\ell \leftarrow 1\) to \(n - 1\) do
3: for each \((u, v) \in E\) do
4: if \(f[u] + w(u, v) < f[v]\) then
5: \(f[v] \leftarrow f[u] + w(u, v)\)
6: return \(f\)

- Issue: when we compute \(f[u] + w(u, v)\), \(f[u]\) may be changed since the end of last iteration
- This is OK: it can only “accelerate” the process!
- After iteration \(\ell\), \(f[v]\) is at most the length of the shortest path from \(s\) to \(v\) that uses at most \(\ell\) edges
- \(f[v]\) is always the length of some path from \(s\) to \(v\)
Bellman-Ford Algorithm

- After iteration $\ell$:
  
  \[
  \text{length of shortest } s-v \text{ path } \leq f[v] \\
  \leq \text{length of shortest } s-v \text{ path using at most } \ell \text{ edges}
  \]

- Assuming there are no negative cycles:
  
  \[
  \text{length of shortest } s-v \text{ path } = \text{length of shortest } s-v \text{ path using at most } n - 1 \text{ edges}
  \]

- So, assuming there are no negative cycles, after iteration $n - 1$:
  
  \[
  f[v] = \text{length of shortest } s-v \text{ path}
  \]
order in which we consider edges: $(s, a)$, $(s, b)$, $(a, b)$, $(a, c)$, $(b, d)$, $(c, d)$, $(d, a)$

end of iteration 1: 0, 2, 7, 2, 4

end of iteration 2: 0, 2, 7, -2, 4

end of iteration 3: 0, 2, 7, -2, 4

Algorithm terminates in 3 iterations, instead of 4.
Bellman-Ford Algorithm

Bellman-Ford\((G, w, s)\)

1: \(f[s] \leftarrow 0\) and \(f[v] \leftarrow \infty\) for any \(v \in V \setminus \{s\}\)
2: for \(\ell \leftarrow 1\) to \(n\) do
3: \(\text{updated} \leftarrow \text{false}\)
4: for each \((u, v) \in E\) do
5: \(\quad\text{if } f[u] + w(u, v) < f[v] \text{ then}\)
6: \(\quad\quad f[v] \leftarrow f[u] + w(u, v), \; \pi[v] \leftarrow u\)
7: \(\quad\text{updated} \leftarrow \text{true}\)
8: if not \(\text{updated}\), then return \(f\)
9: output “negative cycle exists”

- \(\pi[v]\): the parent of \(v\) in the shortest path tree
- Running time = \(O(nm)\)
Outline

1 Minimum Spanning Tree
   • Kruskal’s Algorithm
   • Reverse-Kruskal’s Algorithm
   • Prim’s Algorithm

2 Single Source Shortest Paths
   • Dijkstra’s Algorithm

3 Shortest Paths in Graphs with Negative Weights

4 All-Pair Shortest Paths and Floyd-Warshall

5 Minimum Cost Arborescence
All-Pair Shortest Paths

**Input:** directed graph $G = (V, E)$,
\[ w : E \to \mathbb{R} \text{ (can be negative)} \]

**Output:** shortest path from $u$ to $v$ for every $u, v \in V$

1. **for** every starting point $s \in V$ **do**
2. run Bellman-Ford($G, w, s$)

- Running time $= O(n^2m)$
## Summary of Shortest Path Algorithms we learned

<table>
<thead>
<tr>
<th>algorithm</th>
<th>graph</th>
<th>weights</th>
<th>SS?</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple DP</td>
<td>DAG</td>
<td>$\mathbb{R}$</td>
<td>SS</td>
<td>$O(n + m)$</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>U/D</td>
<td>$\mathbb{R}_{\geq 0}$</td>
<td>SS</td>
<td>$O(n \log n + m)$</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>U/D</td>
<td>$\mathbb{R}$</td>
<td>SS</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>U/D</td>
<td>$\mathbb{R}$</td>
<td>AP</td>
<td>$O(n^3)$</td>
</tr>
</tbody>
</table>

- DAG = directed acyclic graph
- U = undirected
- D = directed
- SS = single source
- AP = all pairs
Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \ldots, n\}$
- For simplicity, extend the $w$ values to non-edges:

$$w(i, j) = \begin{cases} 
0 & i = j \\
\text{weight of edge } (i, j) & i \neq j, (i, j) \in E \\
\infty & i \neq j, (i, j) \notin E 
\end{cases}$$

- For now assume there are no negative cycles

Cells for Floyd-Warshall Algorithm

- First try: $f[i, j]$ is length of shortest path from $i$ to $j$
- Issue: do not know in which order we compute $f[i, j]$’s

$$f^k[i, j]: \text{length of shortest path from } i \text{ to } j \text{ that only uses vertices } \{1, 2, 3, \ldots, k\} \text{ as intermediate vertices}$$
Example for Definition of $f^k[i, j]$’s

\[
\begin{align*}
  f^0[1, 4] &= \infty \\
  f^1[1, 4] &= \infty \\
  f^2[1, 4] &= 140 \quad (1 \rightarrow 2 \rightarrow 4) \\
  f^3[1, 4] &= 90 \quad (1 \rightarrow 3 \rightarrow 2 \rightarrow 4) \\
  f^4[1, 4] &= 90 \quad (1 \rightarrow 3 \rightarrow 2 \rightarrow 4) \\
  f^5[1, 4] &= 60 \quad (1 \rightarrow 3 \rightarrow 5 \rightarrow 4)
\end{align*}
\]
\[ w(i, j) = \begin{cases} 
0 & i = j \\
\text{weight of edge } (i, j) & i \neq j, (i, j) \in E \\
\infty & i \neq j, (i, j) \notin E 
\end{cases} \]

- \( f^k[i, j] \): length of shortest path from \( i \) to \( j \) that only uses vertices \( \{1, 2, 3, \ldots, k\} \) as intermediate vertices

\[ f^k[i, j] = \begin{cases} 
w(i, j) & k = 0 \\
\min \left\{ f^{k-1}[i, j], f^{k-1}[i, k] + f^{k-1}[k, j] \right\} & k = 1, 2, \ldots, n 
\end{cases} \]
Floyd-Warshall($G, w$)

1: $f^0 \leftarrow w$
2: for $k \leftarrow 1$ to $n$ do
3: copy $f^{k-1} \rightarrow f^k$
4: for $i \leftarrow 1$ to $n$ do
5: for $j \leftarrow 1$ to $n$ do
6: if $f^{k-1}[i, k] + f^{k-1}[k, j] < f^k[i, j]$ then
7: $f^k[i, j] \leftarrow f^{k-1}[i, k] + f^{k-1}[k, j]$
Floyd-Warshall\((G, w)\)

1: \(f^{\text{old}} \leftarrow w\)
2: for \(k \leftarrow 1\) to \(n\) do
3: \hspace{0.5cm} copy \(f^{\text{old}} \rightarrow f^{\text{new}}\)
4: for \(i \leftarrow 1\) to \(n\) do
5: \hspace{0.5cm} for \(j \leftarrow 1\) to \(n\) do
6: \hspace{1cm} if \(f^{\text{old}}[i, k] + f^{\text{old}}[k, j] < f^{\text{new}}[i, j]\) then
7: \hspace{1cm} \(f^{\text{new}}[i, j] \leftarrow f^{\text{old}}[i, k] + f^{\text{old}}[k, j]\)

**Lemma** Assume there are no negative cycles in \(G\). After iteration \(k\), for \(i, j \in V\), \(f[i, j]\) is exactly the length of shortest path from \(i\) to \(j\) that only uses vertices in \(\{1, 2, 3, \ldots, k\}\) as intermediate vertices.

- Running time = \(O(n^3)\).
\[ i = 1, \quad i = 2, \quad i = 3, \quad k = 1, \quad k = 2, \quad k = 3, \quad j = 1, \quad j = 2, \quad j = 3, \quad j = 4 \]
Recovering Shortest Paths

Floyd-Warshall($G, w$)

1: $f \leftarrow w$, $\pi[i, j] \leftarrow \perp$ for every $i, j \in V$
2: for $k \leftarrow 1$ to $n$ do
3:   for $i \leftarrow 1$ to $n$ do
4:     for $j \leftarrow 1$ to $n$ do
5:       if $f[i, k] + f[k, j] < f[i, j]$ then
6:         $f[i, j] \leftarrow f[i, k] + f[k, j]$, $\pi[i, j] \leftarrow k$

print-path($i$, $j$)

1: if $\pi[i, j] = \perp$ then then
2:   if $i \neq j$ then print($i$, “,”)
3: else
4:   print-path($i$, $\pi[i, j]$), print-path($\pi[i, j]$, $j$)
Detecting Negative Cycles

**Floyd-Warshall**\((G, w)\)

1: \(f \leftarrow w, \pi[i, j] \leftarrow \perp\) for every \(i, j \in V\)
2: \textbf{for} \(k \leftarrow 1\) \textbf{to} \(n\) \textbf{do}
3: \hspace{1em} \textbf{for} \(i \leftarrow 1\) \textbf{to} \(n\) \textbf{do}
4: \hspace{2em} \textbf{for} \(j \leftarrow 1\) \textbf{to} \(n\) \textbf{do}
5: \hspace{3em} \textbf{if} \(f[i, k] + f[k, j] < f[i, j]\) \textbf{then}
6: \hspace{3em} \(f[i, j] \leftarrow f[i, k] + f[k, j], \pi[i, j] \leftarrow k\)
7: \textbf{for} \(k \leftarrow 1\) \textbf{to} \(n\) \textbf{do}
8: \hspace{1em} \textbf{for} \(i \leftarrow 1\) \textbf{to} \(n\) \textbf{do}
9: \hspace{2em} \textbf{for} \(j \leftarrow 1\) \textbf{to} \(n\) \textbf{do}
10: \hspace{3em} \textbf{if} \(f[i, k] + f[k, j] < f[i, j]\) \textbf{then}
11: \hspace{3em} report “negative cycle exists” and exit
## Summary of Shortest Path Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Graph Type</th>
<th>Weights</th>
<th>SS?</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple DP</td>
<td>DAG</td>
<td>$\mathbb{R}$</td>
<td>SS</td>
<td>$O(n + m)$</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>U/D</td>
<td>$\mathbb{R}_{\geq 0}$</td>
<td>SS</td>
<td>$O(n \log n + m)$</td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>U/D</td>
<td>$\mathbb{R}$</td>
<td>SS</td>
<td>$O(nm)$</td>
</tr>
<tr>
<td>Floyd-Warshall</td>
<td>U/D</td>
<td>$\mathbb{R}$</td>
<td>AP</td>
<td>$O(n^3)$</td>
</tr>
</tbody>
</table>

- **DAG** = directed acyclic graph  
- **U** = undirected  
- **D** = directed  
- **SS** = single source  
- **AP** = all pairs
Outline

1 Minimum Spanning Tree
   - Kruskal’s Algorithm
   - Reverse-Kruskal’s Algorithm
   - Prim’s Algorithm

2 Single Source Shortest Paths
   - Dijkstra’s Algorithm

3 Shortest Paths in Graphs with Negative Weights

4 All-Pair Shortest Paths and Floyd-Warshall

5 Minimum Cost Arborescence
**Def.** An arborescence is directed rooted tree, where all edges are directed away from the root.

### Minimum Cost Arborescence Problem

**Input:** a directed graph \( G = (V, E) \), edge weights \( w : E \to \mathbb{R}_{\geq 0} \), root \( r \in V \)

**Output:** a minimum-cost sub-graph \( T = (V, E') \) of \( G \) that is an arborescence with root \( r \)
Assumptions

- the root $r$ does not have incoming edges.
- every vertex is reachable from the root $r$.

For every $v \in V \setminus \{r\}$, define $l_v = \min_{e \in \delta_v^\text{in}} w(e)$.

For every $v \in V \setminus \{r\}$ and $e \in \delta_v^\text{in}$, define $w'(e) = w(e) - l_v$.

**Lemma** The instances $(G, w, r)$ and $(G, w', r)$ have the same optimum solution.
Lemma  The instances \((G, w, r)\) and \((G, w', r)\) have the same optimum solution.

Proof. Given any tree solution \(T\), \(w(T) - w'(T)\) is always \(\sum_{v \in V \setminus \{r\}} l_v\).

Lemma  Let \((v_0, v_1, v_2, \ldots, v_p = v_0)\) be a cycle \(C\) of 0-cost edges in \(G\). Then there is an optimum solution \(T\), that contains all but one edges in \(C\).
**MCA(G, r, w)**

1: \( F^* \leftarrow \emptyset \)
2: for every \( v \in V \setminus \{r\} \) do
3: \( l_v \leftarrow \min_{e \in \delta_{v}^{in}} w(e) \)
4: for every edge \( e \) entering \( v \) do: \( w'(e) \leftarrow w(e) - l_v \)
5: choose a 0-cost edge entering \( v \), add it to \((V, F^*)\)
6: if \( F^* \) form an arborescence then return \( F^* \)
7: else
8: for every cycle \( C \) in \( F^* \) do: contract \( C \) into a single node
9: let \( G' = (V', E') \) be the obtained graph.
10: \( T' \leftarrow \text{MCA}(G', r, w') \)
11: extend \( T' \) to an aborescence \( T \) in \( G \), by keeping all but one edges in every cycle \( C \) in \( F^* \), and return \( T \)
The running time of the algorithm is $O(mn)$

[Tarjan (1971)]: $O(\min(m \log n, n^2))$

[Gabow, Galil, Spencer, Tarjan (1986)]: $O(n \log n + m)$

[Mendelson, Tarjan, Thorup, Zwick (2006)]: $O(m \log \log n)$