Graph Algorithms

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo
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
   - Bellman-Ford Algorithm

4. All-Pair Shortest Paths and Floyd-Warshall
Spanning Tree

**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 $V$. 

- **Graph** $G = (V, E)$ represents the original graph with vertices $V$ and edges $E$.
- **Spanning Tree** $T = (V, F)$ is a sub-graph of $G$ that forms a tree covering all vertices $V$.

The diagram illustrates a spanning tree $T$ with the selected edges marked in yellow, denoted by $F$, ensuring all vertices $V$ are connected without forming cycles.
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
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
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
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
   - Bellman-Ford Algorithm

4 All-Pair Shortest Paths and Floyd-Warshall
Q: Which edge can be safely included in the MST?
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.
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 $T$
**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 $T$

![Diagram](image-url)
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 $v$
**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 $T'$
**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
Is the Residual Problem Still a MST Problem?

- Residual problem: find the minimum spanning tree that contains edge \((g, h)\)
Residual problem: find the minimum spanning tree that contains edge \((g, h)\)

Contract the edge \((g, h)\)
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 \((u, w)\) \(\in E\), with \(w \neq v\), change it to \((u^*, w)\).

For every edge \((v, w)\) \(\in E\), with \(w \neq u\), change it to \((u^*, w)\).

May create parallel edges! E.g. two edges \((i, g^*)\).
Remove $u$ and $v$ from the graph, and add a new vertex $u^*$
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 \(E\)
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 \((u^*, w)\)
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 \((u^*, w)\)
**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 \((u, w)\) ∈ \(E\), \(w \neq v\), change it to \((u^*, w)\)
- For every edge \((v, w)\) ∈ \(E\), \(w \neq u\), 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
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?
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\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}
Kruskal’s Algorithm: Example

Sets: $\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g\}, \{h\}, \{i\}$
Kruskal’s Algorithm: Example

Sets: \( \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\} \)
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{g, h\}, \{i\}
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f\}, \{g, h\}
Kruskal’s Algorithm: Example

Sets: $\{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}$
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c, i\}, \{d\}, \{e\}, \{f, g, h\}
Kruskal’s Algorithm: Example

Sets: \{a\}, \{b\}, \{c, i, f, g, h\}, \{d\}, \{e\}
Kruskal’s Algorithm: Example

Sets: \( \{a\}, \{b\}, \{c, i, f, g, h\}, \{d\}, \{e\} \)
Kruskal’s Algorithm: Example

Sets: \{a, b\}, \{c, i, f, g, h\}, \{d\}, \{e\}
Sets: \{a, b\}, \{c, i, f, g, h\}, \{d\}, \{e\}
Kruskal’s Algorithm: Example

Sets: \{a, b, c, i, f, g, h\}, \{d\}, \{e\}
Kruskal’s Algorithm: Example

Sets: \{a, b, c, i, f, g, h\}, \{d\}, \{e\}
Kruskal’s Algorithm: Example

Sets: \{a, b, c, i, f, g, h\}, \{d, e\}
Kruskal’s Algorithm: Example

Sets: \{a, b, c, i, f, g, h\}, \{d, e\}
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: \hspace{1em} $S_u \leftarrow$ the set in $S$ containing $u$
6: \hspace{1em} $S_v \leftarrow$ the set in $S$ containing $v$
7: \hspace{1em} if $S_u \neq S_v$ then
8: \hspace{2em} $F \leftarrow F \cup \{(u, v)\}$
9: \hspace{1em} $S \leftarrow S \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$
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 $i$, ($par[i] = \perp$ 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 \( r' \):

\[
\text{par}[r] \leftarrow r'
\]
Q: how can we check if $u$ and $v$ are in the same set?
Q: how can we check if \( u \) and \( v \) are in the same set?
A: Check if root\( (u) = \) root\( (v) \).
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 $u$
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 $r'$: $\text{par}[r] \leftarrow r'$. 
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

\[ \text{root}(v) \]

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

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

Improvement: all vertices in the path directly point to the root, saving time in the future.
Union-Find Data Structure

**root(v)**

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

- Problem: the tree might be too deep; running time might be large
Union-Find Data Structure

root(v)

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

- 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.
Union-Find Data Structure

root(v)

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

root(v)

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

- 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.
**Union-Find Data Structure**

### root($v$)

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

```
3
10 2
12 15 9
7
1 16
13
8
4 11
6
14
5
```

Union-Find Data Structure

root($v$)

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

```
3 10 2
12 15 9
7
1 16
8
4 11
6
14
5
```
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: \hspace{1em} $S_u \leftarrow$ the set in $S$ containing $u$
6: \hspace{1em} $S_v \leftarrow$ the set in $S$ containing $v$
7: \hspace{1em} if $S_u \neq S_v$ then
8: \hspace{2em} $F \leftarrow F \cup \{(u, v)\}$
9: \hspace{2em} $S \leftarrow S \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$
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 $(V, F)$
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 $n \leq 10^{80}$.
MST-Kruskal\((G, w)\)

1: \(F \leftarrow \emptyset\)
2: for every \(v \in V\) do: \(\text{par}[v] \leftarrow \perp\)
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.
**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 $(i, c, f, g)$
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
   - Bellman-Ford Algorithm

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

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.
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.
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?
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.
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
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example

![Graph Diagram]

The graph shows a network with labeled edges and vertices. The vertices are labeled as follows: a, b, c, d, e, h, i, f, g.
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example

Diagram:

- Nodes: a, b, c, d, e, f, g, h, i
- Edges with weights:
  - a to b: 5
  - a to h: 7
  - b to c: 8
  - b to i: 2
  - i to g: 6
  - i to h: 1
  - c to i: 4
  - d to e: 9
  - d to f: 10
  - g to f: 3
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example

Graph with edges and weights:
- a to b: 5
- a to i: 7
- b to c: 8
- b to i: 2
- c to i: 6
- c to g: 4
- d to e: 9
- f to e: 10
- h to i: 1
- h to g: 3
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example

Graph representation:
- Nodes: a, b, c, d, e, i, g, h, f
- Edges with weights:
  - a to b: 5
  - b to c: 8
  - c to i: 2
  - i to g: 6
  - g to f: 3
  - f to e: 10
  - e to d: 9
  - i to h: 7

Red line indicates the edge being removed in the reverse Kruskal’s algorithm.
Reverse Kruskal’s Algorithm: Example
Reverse Kruskal’s Algorithm: Example
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
   - Bellman-Ford Algorithm

4. All-Pair Shortest Paths and Floyd-Warshall
Recall the greedy strategy for Kruskal’s algorithm: choose the edge with the smallest weight.
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 $a$. 
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 $w(T') \leq w(T)$. 


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 $T$
**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 $C$
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 \( C \)
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

Graph with weighted edges:
- a -> b: 5
- a -> h: 12
- a -> i: 11
- b -> c: 8
- c -> d: 13
- c -> i: 2
- d -> e: 9
- d -> f: 14
- f -> e: 10
- h -> g: 6
- g -> i: 7
- g -> f: 3
- i -> g: 1
- i -> c: 4
Prim’s Algorithm: Example
Prims’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example

Graph representation with weighted edges.
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
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. \textbf{while } \(S \neq V\) \textbf{ do}
4. \((u, v) \leftarrow \text{lightest edge between } S \text{ and } V \setminus S, \text{ where } u \in S \text{ and } v \in V \setminus S\)
5. \(S \leftarrow S \cup \{v\}\)
6. \(F \leftarrow F \cup \{(u, v)\}\)
7. \textbf{return } (V, F)
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

1. $d(v) = \min_{u \in S : (u, v) \in E} w(u, v)$: the weight of the lightest edge between $v$ and $S$
2. $\pi(v) = \arg \min_{u \in S : (u, v) \in E} w(u, v)$: $(\pi(v), v)$ 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
Example
Example
Example

(5, a)

(12, a)
Example

(5, a)

(12, a)
Example
Example

\[
\begin{array}{c}
\text{(8, b)} \\
\end{array}
\]
Example
Example
Example
Example
Example
Example
Example
Example
Example

(13, c)
(7, i)
(3, f)
(10, f)
Example
Example

```
(13, c)
(1, g)
(10, f)
```

Graph with nodes labeled a, b, c, d, e, f, g, h, i. Edges labeled with numbers 1 to 14.
Example
Example

\[ (13, c) \]

\[ (1, g) \]

\[ (10, f) \]
Example
Example
Example
Example

(9, e)
Example
Example

\[\begin{array}{c}
\text{Example} \\
\end{array}\]
Example
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

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.

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

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: $u \leftarrow Q$.extract_min()
6: $S \leftarrow S \cup \{u\}$
7: for each $v \in V \setminus S$ such that $(u, v) \in E$ do
8: if $w(u, v) < d(v)$ then
9: $d(v) \leftarrow w(u, v)$, $Q$.decrease_key($v, d(v)$)
10: $\pi(v) \leftarrow u$
11: return $\{(u, \pi(u)) | u \in V \setminus \{s\}\}$
Running Time of Prim’s Algorithm Using Priority Queue

\[ O(n) \times \text{(time for extract\_min)} \ + \ O(m) \times \text{(time for decrease\_key)} \]
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>
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 \(V \setminus U\).
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 \((\{a, b, c, i\}, V \setminus \{a, b, c, i\})\)
**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.

- $e \in \text{MST} \iff$ there is a cut in which $e$ is the lightest edge
- $e \notin \text{MST} \iff$ there is a cycle in which $e$ is the heaviest edge
“Evidence” for $e \in \text{MST}$ or $e \notin \text{MST}$

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

- $e \in \text{MST} \iff$ there is a cut in which $e$ is the lightest edge
- $e \notin \text{MST} \iff$ there is a cycle in which $e$ 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
“Evidence” for $e \in \text{MST}$ or $e \notin \text{MST}$

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

- $e \in \text{MST} \iff$ there is a cut in which $e$ is the lightest edge
- $e \notin \text{MST} \iff$ there is a cycle in which $e$ 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
   - Bellman-Ford Algorithm

4. All-Pair Shortest Paths and Floyd-Warshall
**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 $t$
**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 $t$
**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 or undirected) graph $G = (V, E)$, $s \in V$

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

**Output:** shortest paths from $s$ to all other vertices $v \in V$
## Single Source Shortest Paths

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

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

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

## Reason for Considering Single Source Shortest Paths Problem

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

**Input:** (directed or undirected) 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 or undirected) 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 Problem

- 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 \( G = (V, E) \), \( s \in V \)

\[ w : E \to \mathbb{R}_{\geq 0} \]

**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.
Shortest path from $s$ to $v$ may contain $\Omega(n)$ edges.
Shortest path from $s$ to $v$ may contain $\Omega(n)$ edges

There are $\Omega(n)$ different vertices $v$
- Shortest path from $s$ to $v$ may contain $\Omega(n)$ edges
- There are $\Omega(n)$ different vertices $v$
- Thus, printing out all shortest paths may take time $\Omega(n^2)$
Shortest path from $s$ to $v$ may contain $\Omega(n)$ edges
There are $\Omega(n)$ different vertices $v$
Thus, printing out all shortest paths may take time $\Omega(n^2)$
Not acceptable if graph is sparse
**Shortest Path Tree**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>13</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>12</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>11</td>
<td>12</td>
<td>10</td>
<td>0</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>12</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>13</td>
<td>14</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>
Shortest Path Tree

- \(O(n)\)-size data structure to represent all shortest paths
Shortest Path Tree

- \(O(n)\)-size data structure to represent all shortest paths
- For every vertex \(v\), we only need to remember the parent of \(v\): second-to-last vertex in the shortest path from \(s\) to \(v\) (why?)
Shortest Path Tree

- $O(n)$-size data structure to represent all shortest paths
- For every vertex $v$, we only need to remember the parent of $v$: second-to-last vertex in the shortest path from $s$ to $v$ (why?)
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 $v$

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

A: Breadth first search (BFS) from source $s$
Q: How to compute shortest paths from $s$ when all edges have weight 1?

A: Breadth first search (BFS) from source $s$
Q: How to compute shortest paths from $s$ when all edges have weight 1?

A: Breadth first search (BFS) from source $s$
Q: How to compute shortest paths from $s$ when all edges have weight 1?

A: Breadth first search (BFS) from source $s$
Q: How to compute shortest paths from $s$ when all edges have weight 1?

A: Breadth first search (BFS) from source $s$
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.).
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.

![Diagram](image-url)
**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

```
\begin{center}
\begin{tikzpicture}
\node (u) at (0,0) {$u$};
\node (v) at (1,0) {$v$};
\draw[->] (u) -- node[above] {$4$} (v);
\end{tikzpicture}
\end{center}
```

```
\begin{center}
\begin{tikzpicture}
\node (u) at (0,0) {$u$};
\node (v) at (1,0) {$v$};
\node (w1) at (0.5,0) {$1$};
\node (w2) at (1.5,0) {$1$};
\node (w3) at (2.5,0) {$1$};
\node (w4) at (3.5,0) {$1$};
\draw[->] (u) -- (w1);\draw[->] (w1) -- (w2);\draw[->] (w2) -- (w3);\draw[->] (w3) -- (w4);\draw[->] (w4) -- (v);
\end{tikzpicture}
\end{center}
```

**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
3. $\pi(v) \leftarrow$ vertex from which $v$ is visited
4. $d(v) \leftarrow$ index of the level containing $v$
**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

![Diagram showing an edge (u, v) with weight 4 and a path consisting of 4 unit-weight edges](image)

**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
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!
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.

- Problem: $w(u, v)$ may be too large!

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 $v$
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
Virtual BFS: Example

Time 0
Virtual BFS: Example

Time 2
Virtual BFS: Example

Time 4
Virtual BFS: Example
Virtual BFS: Example

Time 9

Graph:
- Nodes: s, a, b, c, d, e
- Edges and weights:
  - s to a: 4
  - a to b: 5
  - s to c: 2
  - c to d: 5
  - d to e: 4
  - c to a: 3
  - a to d: 4
  - b to e: 3
  - e to d: 7

Time 9
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
   - Bellman-Ford Algorithm

4. All-Pair Shortest Paths and Floyd-Warshall
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 = \(O(n^2)\)
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

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: while $S \neq V$ do
5: \hspace{1em} $u \leftarrow Q\text{.extract\_min()}$
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 $d(u) + w(u, v) < d(v)$ then
9: \hspace{3em} $d(v) \leftarrow d(u) + w(u, v)$, $Q\text{.decrease\_key}(v, d(v))$
10: \hspace{2em} $\pi(v) \leftarrow u$
11: \hspace{1em} 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 \( v \in V: Q.\text{insert}(v, d(v)) \)
4: while \( S \neq V \) do
5: \( u \leftarrow Q.\text{extract\_min}() \)
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: \( d(v) \leftarrow w(u, v), Q.\text{decrease\_key}(v, d(v)) \)
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
   - Bellman-Ford Algorithm

4 All-Pair Shortest Paths and Floyd-Warshall
Recall: Single Source Shortest Path Problem

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 $v \in V$

- Algorithm for the problem: Dijkstra’s algorithm
Dijkstra’s Algorithm Using Priority Queue

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. \(Q \leftarrow \) empty queue, for each \(v \in V\): \(Q.insert(v, d(v))\)
3. while \(S \neq V\) do
5. \(S \leftarrow S \cup \{u\}\)
6. for each \(v \in V \setminus S\) such that \((u, v) \in E\) do
7. if \(d(u) + w(u, v) < d(v)\) then
8. \(d(v) \leftarrow d(u) + w(u, v)\), \(Q.decrease_key(v, d(v))\)
9. \(\pi(v) \leftarrow u\)
10. return \((\pi, d)\)

- Running time = \(O(m + n \log n)\).
Single Source Shortest Paths

**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 \( v \in V \)
Single Source Shortest Paths

**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 \( v \in V \)
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 $v \in V$
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 $v \in V$

- In transition graphs, negative weights make sense
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 $v \in V$

- 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)
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 $v \in V$

- 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
Dijkstra’s Algorithm Fails if We Have Negative Weights
Dijkstra’s Algorithm Fails if We Have Negative Weights
Dijkstra’s Algorithm Fails if We Have Negative Weights
What is the length of the shortest path from $s$ to $d$?

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

Dealing with Negative Cycles
Q: What is the length of the shortest path from $s$ to $d$?
Q: What is the length of the shortest path from $s$ to $d$?

A: $-\infty$
Q: What is the length of the shortest path from $s$ to $d$?

A: $-\infty$
Q: What is the length of the shortest path from $s$ to $d$?

A: $-\infty$
Q: What is the length of the shortest path from \( s \) to \( d \)?

A: \(-\infty\)
Q: What is the length of the shortest path from $s$ to $d$?

A: $-\infty$
Q: What is the length of the shortest path from $s$ to $d$?

A: $-\infty$
Q: What is the length of the shortest path from $s$ to $d$?

A: $-\infty$
Q: What is the length of the shortest path from $s$ to $d$?

A: $-\infty$
Q: What is the length of the shortest path from $s$ to $d$?

A: $-\infty$
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 path from $s$ to $d$?

A: $-\infty$

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

Dealing with Negative Cycles
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.

Dealing with Negative Cycles
- assume the input graph does not contain negative cycles, or
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.

Dealing with Negative Cycles
- assume the input graph does not contain negative cycles, or
- allow algorithm to report “negative cycle exists”
Q: What is the length of the shortest simple path from $s$ to $d$?

A: Unfortunately, computing the shortest simple path between two vertices is an NP-hard problem.
Q: What is the length of the shortest simple path from $s$ to $d$?
Q: What is the length of the shortest simple path from \( s \) to \( d \)?

A: 1
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 **NP-hard** problem.
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
   - Bellman-Ford Algorithm

4. All-Pair Shortest Paths and Floyd-Warshall
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 $v \in V$
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 $v$
### 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
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
\[
f^\ell[v], \, \ell \in \{0, 1, 2, 3, \ldots, n - 1\}, \, v \in V: \text{length of shortest path from } s \text{ to } v \text{ that uses at most } \ell \text{ edges}
\]

\[
f^2[a] =
\]
- \( f^\ell[v], \ \ell \in \{0, 1, 2, 3, \ldots, n-1\}, \ v \in V \): length of shortest path from \( s \) to \( v \) that uses at most \( \ell \) edges
- \( f^2[a] = 6 \)
- $f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots, n-1\}$, $v \in V$: length of shortest path from $s$ to $v$ that uses at most $\ell$ edges
- $f^2[a] = 6$
- $f^3[a] =$
\[ f^\ell[v], \, \ell \in \{0, 1, 2, 3 \cdots , n - 1\}, \, v \in V: \]

length of shortest path from \( s \) to \( v \) that uses at most \( \ell \) edges

- \( f^2[a] = 6 \)
- \( f^3[a] = 2 \)
$f^\ell[v]$, $\ell \in \{0, 1, 2, 3 \cdots , n - 1\}$, $v \in V$ : length of shortest path from $s$ to $v$ that uses at most $\ell$ edges

- $f^2[a] = 6$
- $f^3[a] = 2$

$$f^\ell[v] = \begin{cases} 0 & \ell = 0, v = s \\ \infty & \ell = 0, v \neq s \\ \min \{ f^\ell[u] - 1 \} & \ell > 0 \\ \min \{ (u,v) \in E : (f^\ell[u] + w(u,v)) \} & \ell > 0 \end{cases}$$
\[
f^\ell[v], \ell \in \{0, 1, 2, 3 \cdots, n-1\}, v \in V: \text{length of shortest path from } s \text{ to } v \text{ that uses at most } \ell \text{ edges}
\]

\[
f^2[a] = 6
\]

\[
f^3[a] = 2
\]

\[
f^\ell[v] = \begin{cases} 
0 & \ell = 0, v = s \\
\ell = 0, v \neq s \\
\ell > 0
\end{cases}
\]
\( f^\ell[v], \ell \in \{0, 1, 2, 3 \cdots , n - 1\}, v \in V \): length of shortest path from \( s \) to \( v \) that uses at most \( \ell \) edges

- \( f^2[a] = 6 \)
- \( f^3[a] = 2 \)

\[
f^\ell[v] = \begin{cases} 
0 & \ell = 0, v = s \\
\infty & \ell = 0, v \neq s \\
\min \{ f^\ell[u] - 1[v] \} & \ell > 0 
\end{cases}
\]
\[ f^\ell[v], \; \ell \in \{0, 1, 2, 3, \ldots, n - 1\}, \; v \in V: \]

length of shortest path from \( s \) to \( v \) that uses
at most \( \ell \) edges

- \( f^2[a] = 6 \)
- \( f^3[a] = 2 \)

\[ f^\ell[v] = \begin{cases} 
0 & \ell = 0, v = s \\
\infty & \ell = 0, v \neq s \\
\min \{ & \\
\} & \ell > 0 
\end{cases} \]
\( f^\ell[v] \), \( \ell \in \{0, 1, 2, 3 \cdots, n - 1\} \), \( v \in V \): length of shortest path from \( s \) to \( v \) that uses at most \( \ell \) edges

- \( f^2[a] = 6 \)
- \( f^3[a] = 2 \)

\[
f^\ell[v] = \begin{cases} 
0 & \text{if } \ell = 0, v = s \\
\infty & \text{if } \ell = 0, v \neq s \\
\min \{ & f^{\ell-1}[v] \} & \ell > 0
\end{cases}
\]
• $f^\ell[v]$, \( \ell \in \{0, 1, 2, 3 \cdots , n - 1\} \), \( v \in V \): 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($G, w, s$)

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: if $f^{\ell-1}[u] + w(u, v) < f^\ell[v]$ 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.
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:     if $f^{\ell-1}[u] + w(u, v) < f^\ell[v]$ 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
Dynamic Programming: Example

\[
f^0 \quad s \quad a \quad b \quad c \quad d
\]

\[
\begin{array}{ccccc}
0 & \infty & \infty & \infty & \infty \\
7 & 6 & \cdot & \cdot & \cdot \\
8 & \cdot & \cdot & \cdot & \cdot \\
-4 & -3 & -2 & \cdot & \cdot \\
-2 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]
Dynamic Programming: Example
Dynamic Programming: Example

\[
\begin{array}{cccc}
  & & s & \\
 b & & & \\
 & & a & \\
 c & & & d \\
\end{array}
\]

\[
\begin{array}{cccc}
  & & s & \\
 b & & & \\
 & & a & \\
 c & & & d \\
\end{array}
\]

\[
\begin{array}{cccc}
  & & s & \\
 b & & & \\
 & & a & \\
 c & & & d \\
\end{array}
\]

\[
\begin{array}{cccc}
  & & s & \\
 b & & & \\
 & & a & \\
 c & & & d \\
\end{array}
\]

\[
\begin{array}{cccc}
  & & s & \\
 b & & & \\
 & & a & \\
 c & & & d \\
\end{array}
\]
Dynamic Programming: Example
Dynamic Programming: Example

```
Dynamic Programming: Example

\begin{itemize}
  \item \textbf{Example:} Consider the graph below with weights on the edges.
  \item \textbf{Objective:} Find the shortest path from node \( s \) to node \( d \).
  \item \textbf{Approach:} Use dynamic programming to solve the problem.
  \item Let \( f^0 \) be the function that returns the shortest path from \( s \) to \( c \) and \( f^1 \) be the function that returns the shortest path from \( s \) to \( d \).
  \item The graph is shown below:
  \item \textbf{Solution:} The shortest path from \( s \) to \( d \) is \( s \rightarrow a \rightarrow c \rightarrow d \) with a total weight of -3.
\end{itemize}
```
Dynamic Programming: Example

\[ f^0 \]
\[
\begin{array}{c}
0 \\
6 \\
7 \\
8
\end{array}
\]

\[ f^1 \]
\[
\begin{array}{c}
0 \\
6 \\
7 \\
\infty
\end{array}
\]
Dynamic Programming: Example

\[ f^0 \]

\[ f^1 \]

\[ f^2 \]
Dynamic Programming: Example

\begin{itemize}
\item \textbf{f}^0
\item \textbf{f}^1
\item \textbf{f}^2
\end{itemize}
Dynamic Programming: Example

\[ \begin{align*} f^0: & \quad s & a & b & c & d \\ 0: & \quad 0 & \infty & 8 & \infty & \infty \\ 6: & \quad 6 & 7 & 8 & \infty & \infty \\ -3: & \quad \infty & \infty & -4 & -3 & \infty \\ 7: & \quad \infty & \infty & \infty & -2 & \infty \\
\end{align*} \]
Dynamic Programming: Example

\[ f^0 \]
\[ f^1 \]
\[ f^2 \]
Dynamic Programming: Example
Dynamic Programming: Example

\[
\begin{array}{cccccc}
& & s & & & \\
& 7 & 6 & & & \\
b & & 8 & & a & \\
& & & 8 & 7 & -2 & \\
& & & & b & & c & d \\
& & & & & 7 & -2 & \\
\end{array}
\]

\[
\begin{array}{cccccc}
f^0 & & 0 & & & \\
& & & & 6 & 7 & 8 & -4 & -3 & -2 & \\
f^1 & & 0 & & & \\
& & & 6 & 7 & 8 & -4 & -3 & -2 & 7 & \\
f^2 & & 0 & & & \\
& & & 6 & 7 & 8 & -4 & -3 & -2 & 7 & 2 & 4
\end{array}
\]
Dynamic Programming: Example

Diagram showing the dynamic programming approach for a graph with nodes labeled s, a, b, and c, and edges with weights 7, 6, 8, -3, -4, -2, and 7. The diagram illustrates the progression of values from initial conditions to final values, with labels for $f^0$, $f^1$, $f^2$, and $f^3$.
Dynamic Programming: Example

\[ f^0 \]
\[ s \]
\[ 0 \quad \infty \quad \infty \quad \infty \quad \infty \]
\[ 0 \quad 6 \quad 7 \quad 8 \quad -4 \quad -3 \quad 7 \]
\[ f^1 \]
\[ 0 \quad 6 \quad 7 \quad 8 \quad -4 \quad -3 \quad -2 \]
\[ f^2 \]
\[ 0 \quad 6 \quad 7 \quad 8 \quad -4 \quad -3 \quad 7 \]
\[ f^3 \]
\[ 0 \quad 6 \quad 7 \quad 8 \quad -4 \quad -3 \quad 7 \]
Dynamic Programming: Example

\[ \begin{align*}
&\text{Graph:} \\
&s \rightarrow b, 7, 6, 8, a, 7, 8 \rightarrow c, 7, -4, -3, 7 \rightarrow d, 7, -2, -4, 0
\end{align*} \]
Dynamic Programming: Example

\[ \begin{align*}
\text{s} & \quad 7 & \quad 6 & \\
\text{a} & & & \\
\text{b} & 8 & & \\
\text{c} & & -4 & -3 & 7 \\
\text{d} & & -2 & & \\
\end{align*} \]
Dynamic Programming: Example
Dynamic Programming: Example
Dynamic Programming: Example
Dynamic Programming: Example

Diagrams and values for different functions are shown.
**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. if $f^{\ell-1}[u] + w(u, v) < f^\ell[v]$ 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
**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: if $f^{\ell - 1}[u] + w(u, v) < f^\ell[v]$ 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

**Q:** What if there are negative cycles?
Dynamic Programming With Negative Cycle Detection

dynamic-programming\((G, w, s)\)

1. \(f^0[s] \gets 0\) and \(f^0[v] \gets \infty\) for any \(v \in V \setminus \{s\}\)
2. \textbf{for} \(\ell \gets 1\) to \(n - 1\) \textbf{do}
3. \hspace{1em} copy \(f^{\ell-1} \rightarrow f^\ell\)
4. \hspace{1em} \textbf{for} each \((u, v) \in E\) \textbf{do}
5. \hspace{2em} if \(f^{\ell-1}[u] + w(u, v) < f^\ell[v]\) \textbf{then}
6. \hspace{2.5em} \(f^\ell[v] \gets f^{\ell-1}[u] + w(u, v)\)
7. \hspace{1em} \textbf{for} each \((u, v) \in E\) \textbf{do}
8. \hspace{2em} if \(f^{n-1}[u] + w(u, v) < f^{n-1}[v]\) \textbf{then}
9. \hspace{2.5em} report “negative cycle exists” and exit
10. \textbf{return} \((f^{n-1}[v])_{v \in V}\)
Dynamic Programming with Better Space Usage

\textbf{dynamic-programming}(G, w, s)

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

\(f^\ell\) only depends on \(f^{\ell-1}\): only need 2 vectors
Dynamic Programming with Better Space Usage

dynamic-programming\((G, w, s)\)

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

- \(f^\ell\) only depends on \(f^{\ell-1}\): only need 2 vectors
- only need 1 vector!
Dynamic Programming with Better Space Usage

dynamic-programming\((G, w, s)\)

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:     copy \(f \rightarrow f\)
4:     for each \((u, v) \in E\) do
5:         if \(f[u] + w(u, v) < f[v]\) then
6:             \(f[v] \leftarrow f[u] + w(u, v)\)
7:     copy \(f \rightarrow f\)
8: return \(f\)

- \(f^\ell\) only depends on \(f^{\ell-1}\): only need 2 vectors
- only need 1 vector!
Dynamic Programming with Better Space Usage

dynamic-programming($G, w, s$)

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 $f$

- $f^\ell$ only depends on $f^{\ell-1}$: only need 2 vectors
- only need 1 vector!
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. \(\text{for } \ell \leftarrow 1 \text{ to } n - 1 \text{ do}\)
3. \(\text{for each } (u, v) \in E \text{ do}\)
4. \(\text{if } f[u] + w(u, v) < f[v] \text{ then}\)
5. \(f[v] \leftarrow f[u] + w(u, v)\)
6. \(\text{return } f\)

- \(f^\ell\) only depends on \(f^{\ell-1}\): only need 2 vectors
- only need 1 vector!
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: \textbf{for} \(\ell \leftarrow 1\) to \(n - 1\) \textbf{do}
3: \quad \textbf{for} each \((u, v) \in E\) \textbf{do}
4: \quad \quad \textbf{if} \(f[u] + w(u, v) < f[v]\) \textbf{then}
5: \quad \quad \quad \quad f[v] \leftarrow f[u] + w(u, v)
6: \textbf{return} \(f\)

• Issue: when we compute \(f[u] + w(u, v)\), \(f[u]\) may be changed since the end of last iteration
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: \(\text{for } \ell \leftarrow 1 \text{ to } n - 1 \text{ do}\)
3: \(\text{for each } (u, v) \in E \text{ do}\)
4: \(\text{if } f[u] + w(u, v) < f[v] \text{ then}\)
5: \(f[v] \leftarrow f[u] + w(u, v)\)
6: \(\text{return } f\)

- 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!
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
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

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 $f$

- 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

**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 \(f\)

- 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 \(v\)
- Assuming there are no negative cycles, after iteration \(n - 1\), \(f[v] = \text{length of shortest path from } s \text{ to } v\)
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: \hspace{1em} updated $\leftarrow$ false
4: \hspace{1em} for each $(u, v) \in E$ do
5: \hspace{2em} if $f[u] + w(u, v) < f[v]$ then
6: \hspace{3em} $f[v] \leftarrow f[u] + w(u, v)$
7: \hspace{2em} updated $\leftarrow$ true
8: \hspace{1em} if not updated, then return $f$
9: output “negative cycle exists”
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: if \(f[u] + w(u, v) < f[v]\) then
6: \(f[v] \leftarrow f[u] + w(u, v), \pi[v] \leftarrow u\)
7: \(\text{updated} \leftarrow \text{true}\)
8: if not \(\text{updated}\), then return \(f\)
9: output “negative cycle exists”

\(\pi[v]\): the parent of \(v\) in the shortest path tree
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: \textbf{for } \ell \leftarrow 1 \textbf{ to } n \textbf{ do}
3: \hspace{1em} updated \leftarrow \text{false}
4: \textbf{for each } (u, v) \in E \textbf{ do}
5: \hspace{2em} \textbf{if } f[u] + w(u, v) < f[v] \textbf{ then}
6: \hspace{3em} f[v] \leftarrow f[u] + w(u, v), \pi[v] \leftarrow u
7: \hspace{2em} updated \leftarrow \text{true}
8: \hspace{1em} \textbf{if not } updated, \textbf{ then return } f
9: \text{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
   • Bellman-Ford Algorithm

4 All-Pair Shortest Paths and Floyd-Warshall
### 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>
</tbody>
</table>

- DAG = directed acyclic graph     U = undirected     D = directed
- SS = single source     AP = all pairs
## 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>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
All-Pair Shortest Paths

Input: directed graph $G = (V, E)$,

$w : E \rightarrow \mathbb{R}$ (can be negative)

Output: shortest path from $u$ to $v$ for every $u, v \in V$
# All-Pair Shortest Paths

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

$$w : E \rightarrow \mathbb{R} \text{ (can be negative)}$$

**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($G, w, s$)
All-Pair Shortest Paths

**Input:** directed graph $G = (V, E)$, $w : E \to \mathbb{R}$ (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 $= O(n^2m)$
Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
It is convenient to assume \( V = \{1, 2, 3, \cdots, n\} \)

For simplicity, extend the \( w \) values to non-edges:

\[
w(i, j) = \begin{cases} 
0 & \text{if } i = j \\
\text{weight of edge } (i, j) & \text{if } i \neq j, (i, j) \in E \\
\infty & \text{if } i \neq j, (i, j) \notin E
\end{cases}
\]
Design a Dynamic Programming Algorithm

- It is convenient to assume \( V = \{1, 2, 3, \cdots, n\} \)
- 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
It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$

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
Design a Dynamic Programming Algorithm

- It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$
- 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 $j$
**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
It is convenient to assume $V = \{1, 2, 3, \cdots, n\}$

For simplicity, extend the $w$ values to non-edges:

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

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

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

- \( f^k[i, j] \): length of shortest path from \( i \) to \( j \) that only uses vertices \( \{1, 2, 3, \ldots, k\} \) as intermediate vertices
\[ w(i, j) = \begin{cases} 
0 & \text{if } i = j \\
\text{weight of edge } (i, j) & \text{if } i \neq j, (i, j) \in E \\
\infty & \text{if } i \neq j, (i, j) \notin E
\end{cases} \]

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

\[ f^k[i, j] = \begin{cases} 
k = 0 \\
k = 1, 2, \cdots, n
\end{cases} \]
\[
    w(i, j) = \begin{cases} 
        0 & \text{if } i = j \\
        \text{weight of edge } (i, j) & \text{if } i \neq j, (i, j) \in E \\
        \infty & \text{if } i \neq j, (i, j) \notin E
    \end{cases}
\]

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

\[
    f^k[i, j] = \begin{cases} 
        w(i, j) & k = 0 \\
        \min \{ f^{k-1}[i, j], f^{k-1}[i, k] + f^{k-1}[k, j] \} & k = 1, 2, \ldots, n
    \end{cases}
\]
\[ w(i, j) = \begin{cases} 
0 & i = j \\
\text{weight of edge } (i, j) & i \neq j, (i, j) \in E \\
\infty & i \neq j, (i, j) \notin E 
\end{cases} \]

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

\[ f^k[i, j] = \begin{cases} 
w(i, j) & k = 0 \\
\min \{ \} & k = 1, 2, \cdots, n 
\end{cases} \]
$$w(i, j) = \begin{cases} 0 & i = j \\ \text{weight of edge } (i, j) & i \neq j, (i, j) \in E \\ \infty & i \neq j, (i, j) \notin E \end{cases}$$

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

$$f^k[i, j] = \begin{cases} w(i, j) & k = 0 \\ \min \left\{ f^k[i, j], f^{k-1}[i, j] \right\} & k = 1, 2, \ldots, n \end{cases}$$
\[ w(i, j) = \begin{cases} 
0 & i = j \\
\text{weight of edge } (i, j) & i \neq j, (i, j) \in E \\
\infty & i \neq j, (i, j) \notin E 
\end{cases} \]

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

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 = $O(n^3)$. 
Floyd-Warshall \((G, w)\)

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

**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** \(= O(n^3)\).
Floyd-Warshall($G, w$)

1: $f \leftarrow w$
2: for $k \leftarrow 1$ to $n$ do
3: copy $f \rightarrow f$
4: for $i \leftarrow 1$ to $n$ do
5: for $j \leftarrow 1$ to $n$ do
6: if $f[i, k] + f[k, j] < f[i, j]$ then
7: $f[i, j] \leftarrow f[i, k] + f[k, j]$

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 $= O(n^3)$. 
Floyd-Warshall($G, w$)

1: $f \leftarrow w$
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:                 $f[i, j] \leftarrow f[i, k] + f[k, j]$

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, ..., $k$\} as intermediate vertices.

Running time $= O(n^3)$. 
**Floyd-Warshall**$(G, w)$

1: $f \leftarrow w$
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:                 $f[i, j] \leftarrow f[i, k] + f[k, j]$

**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 $\{1, 2, 3, \cdots, k\}$ as intermediate vertices.

$\text{Running time} = O(n^3)$. 

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

1: \( f \leftarrow w \)  
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: \( f[i, j] \leftarrow f[i, k] + f[k, j] \)

**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 \( \{1, 2, 3, \cdots, k\} \) as intermediate vertices.

- Running time = \( O(n^3) \).
Floyd-Warshall($G, w$)

1: $f \leftarrow w$, $\pi[i, j] \leftarrow \bot$ 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 $\pi[i, j] = \bot$ then
2:     if $i \neq j$ then
3:         print($i, \"\", j$)
4:     else
5:         print-path($i, \pi[i, j]$), print-path($\pi[i, j], j$)
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
2. if $i \neq j$ then print($i, "\"", j$)
3. else
4. print-path($i, \pi[i, j]$), print-path($\pi[i, j], j$)
Detecting Negative Cycles

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

1. \( f \leftarrow w, \pi[i, j] \leftarrow \bot \) 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. \[ \text{if } f[i, k] + f[k, j] < f[i, j] \text{ then} \]
6. \[ f[i, j] \leftarrow f[i, k] + f[k, j], \pi[i, j] \leftarrow k \]
Detecting Negative Cycles

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: $f[i, j] \leftarrow f[i, k] + f[k, j]$, $\pi[i, j] \leftarrow k$
7: for $k \leftarrow 1$ to $n$ do
8: for $i \leftarrow 1$ to $n$ do
9: for $j \leftarrow 1$ to $n$ do
10: if $f[i, k] + f[k, j] < f[i, j]$ then
11: report “negative cycle exists” and exit
## Summary of Shortest Path Algorithms

<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