An Activity-Selection Problem

Suppose:

- You are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of $n$ proposed activities that wish to use a resource, such as a lecture hall, which can serve only one activity at a time.
- Each activity $a_i$ has a start time $s_i$ and finish time $f_i$, where $0 \leq s_i < f_i < \infty$. If selected, activity $a_i$ takes place during the half-open time interval $[s_i, f_i)$.
- Activities $a_i$ and $a_j$ are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, $a_i$ and $a_j$ are compatible if $s_i \geq f_j$ or $s_j \geq f_i$.

**Goal:** Select a maximum-size subset of mutually compatible activities.

Assume that the activities are sorted in monotonically increasing order of finish time: $f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$. 
An Activity-Selection Problem

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

$a_1$

$a_2$

$a_3$

$a_4$

$a_5$

$a_6$

$a_7$

$a_8$

$a_9$

$a_{10}$

$a_{11}$
An Activity-Selection Problem

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

A mutually compatible set of activities
An Activity-Selection Problem

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

A largest mutually compatible set of activities
An Activity-Selection Problem

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

Another largest mutually compatible set of activities
**Activity-Selection: Optimal Substructure**

Let $S_{ij}$ = set of activities that start after $a_i$ finishes and finishes before $a_j$ starts

$A_{ij} = a$ maximum set of mutually compatible activities in $S_{ij}$, which includes some activity $a_k$

Now by including $a_k$ in an optimal solution we are left with the following two subproblems:

- finding mutually compatible activities in $S_{ik}$
- finding mutually compatible activities in $S_{kj}$

Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$.

Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ and $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.

The cut-and-paste argument shows that the optimal solution $A_{ij}$ must also include optimal solutions to subproblems for $S_{ik}$ and $S_{kj}$.
Activity-Selection: Recurrence Relation

We have, \( A_{ij} = A_{ik} \cup \{ a_k \} \cup A_{kj} \) and \( |A_{ij}| = |A_{ik}| + |A_{kj}| + 1 \).

Let \( c[i, j] = \) size of an optimal solution for the set \( S_{ij} \).

Then

\[
c[i, j] = \begin{cases} 
0, & \text{if } S_{ij} = \emptyset, \\
\max_{a_k \in S_{ij}} \{ c[i, k] + c[k, j] + 1 \}, & \text{if } S_{ij} \neq \emptyset.
\end{cases}
\]

Hence, we can use either recursion with memorization or bottom-up dynamic programming to solve the problem in \( \Theta(n^3) \) time.

Can we do better?
Activity-Selection: Improvement (Greedy Choice)

\[
c[i, j] = \begin{cases} 
0, & \text{if } S_{ij} = \emptyset, \\
\max_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\}, & \text{if } S_{ij} \neq \emptyset.
\end{cases}
\]

Instead of iterating over all \(a_k \in S_{ij}\) and checking solutions to subproblems for \(S_{ik}\) and \(S_{kj}\) to find the optimal \(a_k\), can we find the optimal \(a_k\) without even solving the subproblems?

Observe that among the activities we choose for our solution, one must be the first one to finish. Intuitively, therefore, we should choose the activity in the input with the earliest finish time, since that would leave the resource available for as many of the activities that follow it as possible.
Activity-Selection: Improvement (Greedy Choice)

Let’s consider choosing the activity in the input with the earliest finish time.

Since the activities set in the input $S = \{a_1, a_2, ..., a_n\}$ sorted in monotonically increasing order of finish time, i.e., $f_1 \leq f_2 \leq f_3 \leq ... \leq f_{n-1} \leq f_n$, we should choose $a_1$ to be in our solution.

Let $S_k = \{a_i \in S|s_i \geq f_k\}$, i.e., the set of activities that start after activity $a_k$ finishes.

If we make the greedy choice of activity $a_1$, then $S_1$ remains as the only subproblem to solve.

Optimal substructure tells us that if $a_1$ is in the optimal solution, then an optimal solution to the original problem consists of activity $a_1$ and all the activities in an optimal solution to the subproblem $S_1$.

But is the intuition correct?
**Activity-Selection: Improvement (Greedy Choice)**

**Theorem:** Consider any nonempty subproblem $S_k$, and let $a_m$ be an activity in $S_k$ with the earliest finish time. Then $a_m$ is included in some maximum-size subset of mutually compatible activities of $S_k$.

**Proof:** Let $A_k = \text{a maximum-size subset of mutually compatible activities in } S_k$.

Let $a_j$ be the activity in $A_k$ with the earliest finish time.

If $a_j = a_m$, we are done, since we have shown that $a_m$ is in some maximum-size subset of mutually compatible activities of $S_k$.

If $a_j \neq a_m$, let $A_k' = A_k - \{a_j\} \cup \{a_m\}$.

The activities in $A_k'$ are disjoint because the activities in $A_k$ are disjoint, $a_j$ is the first activity in $A_k$ to finish, and $f_m \leq f_j$.

Since $|A_k'| = |A_k|$, we conclude that $A_k'$ is a maximum-size subset of mutually compatible activities of $S_k$, and it includes $a_m$. 

11
Activity-Selection: Recursive Algorithm

**Recursive-Activity-Selector** \((s, f, k, n)\)

1. \(m \leftarrow k + 1\)
2. **while** \(m \leq n\) **and** \(s[m] < f[k]\)
3. \(m \leftarrow m + 1\)
4. **if** \(m \leq n\) **then**
5. \(\text{return } \{a_m\} \cup \text{Recursive-Activity-Selector} (s, f, m, n)\)
6. **else** \(\text{return } \emptyset\)
Greedy Activity Selection

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

Diagram showing the activities $a_1$ to $a_{11}$ with their respective start and finish times.
**Greedy Activity Selection**

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>
Greedy Activity Selection

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>
Greedy Activity Selection

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

---

Diagram showing the activities $a_1$ to $a_{11}$ with their start and finish times.
Greedy Activity Selection

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>
Greedy Activity Selection

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

Diagram showing the activities $a_1$ to $a_{11}$ with start and finish times.
Greedy Activity Selection

An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

Diagram showing the activities $a_1$ to $a_{11}$ with their start times $s_i$ and finish times $f_i$.
## Greedy Activity Selection

### An example set $S$ of activities

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$f_i$</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16

- $a_1$
- $a_2$
- $a_3$
- $a_4$
- $a_5$
- $a_6$
- $a_7$
- $a_8$
- $a_9$
- $a_{10}$
- $a_{11}$
Activity-Selection: Iterative Algorithm

\[ \text{GREEDY}\text{ACTIVITY-SECTOR} (s, f) \]

1. \( n \leftarrow \text{s.length} \)
2. \( A \leftarrow \{a_1\} \)
3. \( k \leftarrow 1 \)
4. \text{for } m \text{ from 2 to } n \text{ do}
5. \quad \text{if } s[m] \geq f[k] \text{ then}
6. \quad A \leftarrow A \cup \{a_m\}
7. \quad k \leftarrow m
8. \text{return } A

Running time = \( \Theta(n) \)
The Minimum Spanning Tree (MST) Problem

We are given a weighted connected undirected graph $G = (V, E)$ with vertex set $V$ and edge set $E$, and a weight function $w$ such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

Our goal is to find an acyclic subset $T \subseteq E$ that connects all vertices of $V$ and whose total weight $w(T) = \sum_{(u, v) \in T} w(u, v)$ is minimized.

Since $T$ is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it “spans” the graph $G$.

We call the problem of determining the tree $T$ the minimum-spanning-tree problem.
The Minimum Spanning Tree (MST) Problem

A weighted undirected graph
The Minimum Spanning Tree (MST) Problem

A weighted undirected graph

Its MST (in red) of total weight 37
MST: Greedy Strategy for Growing an MST

We are given a weighted connected undirected graph $G = (V, E)$ with vertex set $V$ and edge set $E$, and a weight function $w$ such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

Suppose set $A \subset E$ is a subset of some MST of $G$.

Now if edge $(u, v) \in E$ but edge $(u, v) \notin A$, we call $(u, v)$ a safe edge provided $A \cup \{u, v\}$ is also a subset of an MST of $G$. 
MST: Greedy Strategy for Growing an MST

\[ \text{Generic-MST} \left( G = (V, E), w \right) \]

1. \( A \leftarrow \emptyset \)
2. \( \text{while } A \text{ does not form a spanning tree of } G \text{ do} \)
3. \( \text{find an edge } (u, v) \in E \text{ that is safe for } A \)
4. \( A \leftarrow A \cup \{(u, v)\} \)
5. \( \text{return } A \)
**MST: Finding Safe Edges**

A cut \((S, V \setminus S)\) of an undirected graph \(G = (V, E)\) is a partition of \(V\).

We say that an edge \((u, v) \in E\) crosses the cut \((S, V \setminus S)\) if one of its endpoints is in \(S\) and the other is in \(V \setminus S\).

We say that a cut respects a set \(A\) of edges if no edge in \(A\) crosses the cut.

An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut.

Note that there can be more than one light edge crossing a cut in the case of ties.

More generally, we say that an edge is a light edge satisfying a given property if its weight is the minimum of any edge satisfying the property.
Green vertices belong to set $S$, i.e., $S = \{a, b, d, e\}$.
White vertices belong to set $V - S$, i.e., $V - S = \{c, f, g, h, i\}$.
The red line represent the cut $(S, V - S)$.
Dotted edges are the cut edges, i.e., they cross the red line.
Blue thick edges form set $A$, i.e.,
$$A = \{(a, b), (c, f), (c, i), (f, g), (g, h)\}.$$
**MST: Finding Safe Edges**

**Theorem:** Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function $w$ defined on $E$. Let $A$ be a subset of $E$ that is included in some minimum spanning tree for $G$, and let $(S, V \setminus S)$ be any cut of $G$ that respects $A$, and let $(u, v)$ be a light edge crossing $(S, V \setminus S)$. Then, edge $(u, v)$ is safe for $A$.

**Corollary:** Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function $w$ defined on $E$. Let $A$ be a subset of $E$ that is included in some minimum spanning tree for $G$, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$ if $(u, v)$ is a light edge crossing $C$ to some other component of $G_A$, then edge $(u, v)$ is safe for $A$. 
A Disjoint-Set Data Structure (for Kruskal’s MST Algorithm)

A disjoint-set data structure maintains a collection of disjoint dynamic sets. Each set is identified by a representative which must be a member of the set.

The collection is maintained under the following operations:

**MAKE-SET( x )**: create a new set \{x\} containing only element \(x\).
Element \(x\) becomes the representative of the set.

**FIND( x )**: returns a pointer to the representative of the set containing \(x\)

**UNION( x, y )**: replace the dynamic sets \(S_x\) and \(S_y\) containing \(x\) and \(y\), respectively, with the set \(S_x \cup S_y\)
A Disjoint-Set Data Structure (for Kruskal’s MST Algorithm)

**Make-Set (x)**
1. $\pi(x) \leftarrow x$
2. $\text{rank}(x) \leftarrow 0$

**Link (x, y)**
1. if $\text{rank}(x) > \text{rank}(y)$ then $\pi(y) \leftarrow x$
2. else $\pi(x) \leftarrow y$
3. if $\text{rank}(x) = \text{rank}(y)$ then $\text{rank}(y) \leftarrow \text{rank}(y) + 1$

**Union (x, y)**
1. **Link (Find (x), Find (y))**

**Find (x)**
1. if $x \neq \pi(x)$ then $\pi(x) \leftarrow \text{Find} (\pi(x))$
2. return $\pi(x)$
**A Disjoint-Set Data Structure**
*(for Kruskal’s MST Algorithm)*

**Theorem:** A sequence of $N$ **MAKE-SET**, **UNION** and **FIND** operations of which exactly $n \leq N$ are **MAKE-SET** operations takes $O(N\alpha(n))$ times to execute, where $\alpha(n)$ is the extremely slowly growing *Inverse Ackermann Function* which has a value no larger than 3 for all practical values of $n$.

**Proof:** We will prove this later in the semester.
MST: Kruskal’s Algorithm

\[
\text{MST-Kruskal} \ (G = (V, E), w)
\]

1. \( A \leftarrow \emptyset \)
2. \( \text{for each vertex } v \in G.V \text{ do} \)
3. \( \text{MAKE-SET}(v) \)
4. \( \text{sort the edges of } G.E \text{ into nondecreasing order by weight } w \)
5. \( \text{for each edge } (u, v) \in G.E \text{ taken in nondecreasing order by weight do} \)
6. \( \text{if } \text{FIND-SET}(u) \neq \text{FIND-SET}(v) \text{ then} \)
7. \( A \leftarrow A \cup \{(u, v)\} \)
8. \( \text{UNION}(u, v) \)
9. \( \text{return } A \)
MST: Kruskal’s Algorithm

Initial State

![Graph with labeled edges and vertices representing an initial state of a minimum spanning tree algorithm.]

- Edges: (a,b) 4, (a,h) 8, (h,i) 7, (i,g) 6, (g,f) 2, (f,d) 9, (d,c) 7, (b,8) 8, (c,i) 2, (c,d) 4, (i,e) 14, (e,f) 10.
MST: Kruskal’s Algorithm

(1) edge \((h, g)\)
MST: Kruskal’s Algorithm

(1) edge \((h, g)\)
MST: Kruskal’s Algorithm

(2) edge \((i, c)\)
MST: Kruskal’s Algorithm

(2) edge \((i, c)\)
MST: Kruskal’s Algorithm

(3) edge (g, f)
MST: Kruskal’s Algorithm

(3) edge \((g, f)\)
MST: Kruskal’s Algorithm

(4) edge \((a, b)\)
MST: Kruskal’s Algorithm

(4) edge \((a, b)\)
MST: Kruskal’s Algorithm

(5) edge \((c, f)\)
MST: Kruskal’s Algorithm

(5) edge \((c, f)\)
MST: Kruskal’s Algorithm

(6) edge $\langle i, g \rangle$
MST: Kruskal’s Algorithm

(6) edge \((i, g)\)
MST: Kruskal’s Algorithm

(7) edge (c, d)
MST: Kruskal’s Algorithm

(7) edge $(c, d)$
MST: Kruskal’s Algorithm

(8) edge \((i, h)\)
MST: Kruskal’s Algorithm

(8) edge $(i, h)$
MST: Kruskal’s Algorithm

(9) edge \((a, h)\)
MST: Kruskal’s Algorithm

(9) edge \((a, h)\)
MST: Kruskal’s Algorithm

(10) edge (b, c)
MST: Kruskal’s Algorithm

(10) edge \((b, c)\)
MST: Kruskal’s Algorithm

(11) edge \((d, e)\)
MST: Kruskal’s Algorithm

(11) edge \((d, e)\)
MST: Kruskal’s Algorithm

(12) edge \((e, f)\)
MST: Kruskal’s Algorithm

(12) edge \((e, f)\)
MST: Kruskal’s Algorithm

(13) edge \((b, h)\)
MST: Kruskal’s Algorithm

(13) edge $(b, h)$
MST: Kruskal’s Algorithm

(14) edge ($d, f$)
MST: Kruskal’s Algorithm

(14) edge \((d, f)\)
MST: Kruskal’s Algorithm

(14) edge \((d, f)\)

Total weight = 37
MST: Kruskal’s Algorithm

\[ \text{MST-Kruskal} \ ( G = (V, E), \ w ) \]

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

Let \( n = |V| \) and \( m = |E| \). Since \( G \) is connected, we have \( m \geq n - 1 \). Then the sorting in step 4 can be done in \( O(m \log m) \) time.

\#disjoint-set operations performed, \( N = 2m + 2n - 1 \), of which
\[ \#\text{MAKE-SET: } n, \quad \#\text{FIND-SET: } 2m, \quad \#\text{UNION: } n - 1 \]

So, total time taken by disjoint-set operations = \( O((n + m)\alpha(n)) \)

Hence, MST-Kruskal’s running time = \( O(m \log m) \)
MST: Prim’s Algorithm

\[ MST-Prim \ ( G = (V, E), \ w, \ r ) \]

1. for each vertex \( v \in G.V \) do
2. \( v.d \leftarrow \infty \)
3. \( v.\pi \leftarrow NIL \)
4. \( r.d \leftarrow 0 \)
5. Min-Heap \( Q \leftarrow \emptyset \)
6. for each vertex \( v \in G.V \) do
7. \( \text{INSERT}( Q, v ) \)
8. while \( Q \neq \emptyset \) do
9. \( u \leftarrow \text{EXTRACT-MIN}( Q ) \)
10. for each \( (u,v) \in G.E \) do
11. \( \text{if } v \in Q \text{ and } w(u,v) < v.d \text{ then} \)
12. \( v.d \leftarrow w(u,v) \)
13. \( v.\pi \leftarrow u \)
14. \( \text{DECREASE-KEY}( Q, v, w(u,v) ) \)
MST: Prim’s Algorithm

Initial State

Graph with labeled edges and vertices.
MST: Prim’s Algorithm

Step 1: add vertex $a$ to MST
Step 1’: update neighbors of \( a \)
MST: Prim’s Algorithm

Step 2: add vertex $b$ through edge $(a, b)$
MST: Prim’s Algorithm

Step 2’: update neighbors of $b$
MST: Prim’s Algorithm

Step 3: add vertex $c$ through edge $(b, c)$
MST: Prim’s Algorithm

Step 3’: update neighbors of \( c \)
Step 4: add vertex $i$ through edge $(c, i)$
MST: Prim’s Algorithm

Step 4’: update neighbors of $i$
MST: Prim’s Algorithm

Step 5: add vertex $f$ through edge $(c, f)$
Step 5′: update neighbors of $f$
Step 6: add vertex $g$ through edge $(f, g)$
MST: Prim’s Algorithm

Step 6’: update neighbors of g
Step 7: add vertex $h$ through edge $(g, h)$
MST: Prim’s Algorithm

Step 7’: update neighbors of $h$
Step 8: add vertex $d$ through edge $(c, d)$
MST: Prim’s Algorithm

Step 8’: update neighbors of d
Step 9: add vertex $e$ through edge $(d, e)$
Step 9': update neighbors of $e$
MST: Prim’s Algorithm

Done

Total weight = 37
MST: Prim’s Algorithm

Let $n = |V|$ and $m = |E|$.

# INSERTS = $n$

# EXTRACT-MINS = $n$

# DECREASE-KEYS $\leq m$

Total cost

\[ \leq n \left( \text{cost}_{\text{Insert}} + \text{cost}_{\text{Extract-Min}} \right) + m \left( \text{cost}_{\text{Decrease-Key}} \right) \]
MST: Prim’s Algorithm

MST-Prim \((G = (V, E), w, r)\)

1. for each vertex \(v \in G.V\) do
2. \(v.d \leftarrow \infty\)
3. \(v.\pi \leftarrow \text{NIL}\)
4. \(r.d \leftarrow 0\)
5. Min-Heap \(Q \leftarrow \emptyset\)
6. for each vertex \(v \in G.V\) do
7. \(\text{INSERT}(Q, v)\)
8. while \(Q \neq \emptyset\) do
9. \(u \leftarrow \text{EXTRACT-MIN}(Q)\)
10. for each \((u, v) \in G.E\) do
11. if \(v \in Q\) and \(w(u, v) < v.d\) then
12. \(v.d \leftarrow w(u, v)\)
13. \(v.\pi \leftarrow u\)
14. \(\text{DECREASE-KEY}(Q, v, w(u, v))\)

Let \(n = |V|\) and \(m = |E|\)

For Binary Heap (worst-case costs):

\(\text{cost}_{\text{Insert}} = O(\log n)\)
\(\text{cost}_{\text{Extract-Min}} = O(\log n)\)
\(\text{cost}_{\text{Decrease-Key}} = O(\log n)\)

\[\therefore \text{Total cost (worst-case)} = O((m + n) \log n)\]
MST: Prim’s Algorithm

\begin{center}
\textbf{MST-Prim ( \( G = (V,E) \), \( w \), \( r \) )}
\end{center}

1. \textbf{for each vertex} \( v \in G.V \) \textbf{do}
2. \hspace{0.5cm} \( v.d \leftarrow \infty \)
3. \hspace{0.5cm} \( v.\pi \leftarrow NIL \)
4. \hspace{0.5cm} \( r.d \leftarrow 0 \)
5. \hspace{0.5cm} \textbf{Min-Heap} \( Q \leftarrow \emptyset \)
6. \textbf{for each vertex} \( v \in G.V \) \textbf{do}
7. \hspace{0.5cm} \textbf{INSERT} \( (Q,v) \)
8. \textbf{while} \( Q \neq \emptyset \) \textbf{do}
9. \hspace{1.0cm} \( u \leftarrow \textbf{EXTRACT-MIN}(Q) \)
10. \hspace{1.5cm} \textbf{for each} \( (u,v) \in G.E \) \textbf{do}
11. \hspace{2.0cm} \textbf{if} \( v \in Q \) \textbf{and} \( w(u,v) < v.d \) \textbf{then}
12. \hspace{2.5cm} \( v.d \leftarrow w(u,v) \)
13. \hspace{2.5cm} \( v.\pi \leftarrow u \)
14. \hspace{2.0cm} \textbf{DECREASE-KEY} \( (Q,v,w(u,v)) \)

Let \( n = |V| \) and \( m = |E| \)

For Fibonacci Heap ( amortized ):

\[
\begin{align*}
\text{cost}_{\text{Insert}} &= O(1) \\
\text{cost}_{\text{Extract-Min}} &= O(\log n) \\
\text{cost}_{\text{Decrease-Key}} &= O(1)
\end{align*}
\]

\( \therefore \) Total cost ( amortized )
\[
= O(m + n \log n)
\]
The Single-Source Shortest Paths (SSSP) Problem

We are given a weighted, directed graph $G = (V, E)$ with vertex set $V$ and edge set $E$, and a weight function $w$ such that for each edge $(u, v) \in E$, $w(u, v)$ represents its weight.

We are also given a source vertex $s \in V$.

Our goal is to find a shortest path (i.e., a path of the smallest total edge weight) from $s$ to each vertex $v \in V$. 
**SSSP: Relaxation**

Initialize-Single-Source \( (G = (V, E), s) \)

1. for each vertex \( v \in G.V \) do
2. \( v.d \leftarrow \infty \)
3. \( v.\pi \leftarrow \text{NIL} \)
4. \( s.d \leftarrow 0 \)

Relax \( (u, v, w) \)

1. if \( u.d + w(u, v) < v.d \) then
2. \( v.d \leftarrow u.d + w(u, v) \)
3. \( v.\pi \leftarrow u \)
SSSP: Properties of Shortest Paths and Relaxation

The **weight** $w(p)$ of path $p = \langle v_0, v_1, ..., v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the **shortest-path weight** $\delta(u, v)$ from $u$ to $v$ by

$$\delta(u, v) = \begin{cases} \min \{w(p): p \text{ is } u \sim v\}, & \text{if there is a path from } u \text{ to } v, \\ \infty, & \text{otherwise.} \end{cases}$$

A **shortest path** from vertex $u$ to vertex $v$ is then defined as any path $p$ with weight $w(p) = \delta(u, v)$. 
SSSP: Properties of Shortest Paths and Relaxation

**Triangle inequality** (Lemma 24.10 of CLRS)

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

**Upper-bound inequality** (Lemma 24.11 of CLRS)

We always have \(v.d \geq \delta(s, v)\) for all vertices \(v \in V\), and once \(v.d\) achieves the value \(\delta(u, v)\), it never changes.

**No-path property** (Corollary 24.12 of CLRS)

If there is no path from \(s\) to \(v\), then we always have \(v.d = \delta(s, v) = \infty\).

**Convergence property** (Lemma 24.14 of CLRS)

If \(s \leadsto u \rightarrow v\) is a shortest path in \(G\) for some \(u, v \in V\), and if \(u.d = \delta(s, u)\) at any time prior to relaxing edge \((u, v)\), then \(v.d = \delta(s, v)\) at all times afterward.
SSSP: Properties of Shortest Paths and Relaxation

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

Predecessor-subgraph property (Lemma 24.17 of CLRS)
Once $v. d = \delta(s, v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at $s$. 
**Dijkstra’s SSSP Algorithm with a Min-Heap**  
( SSSP: Single-Source Shortest Paths )

**Input:** Weighted graph $G = (V, E)$ with vertex set $V$ and edge set $E$, a weight function $w$, and a source vertex $s \in G[V]$.

**Output:** For all $v \in G[V]$, $v.d$ is set to the shortest distance from $s$ to $v$.

---

**Dijkstra-SSSP** ($G = (V, E)$, $w$, $s$)

1. for each vertex $v \in G.V$ do
2. \hspace{1em} $v.d \leftarrow \infty$
3. \hspace{1em} $v.\pi \leftarrow NIL$
4. \hspace{1em} $s.d \leftarrow 0$
5. \hspace{1em} Min-Heap $Q \leftarrow \emptyset$
6. for each vertex $v \in G.V$ do
7. \hspace{1em} INSERT($Q$, $v$)
8. while $Q \neq \emptyset$ do
9. \hspace{1em} $u \leftarrow \text{EXTRACT-MIN}(Q)$
10. for each $(u, v) \in G.E$ do
11. \hspace{1em} if $u.d + w(u, v) < v.d$ then
12. \hspace{2em} $v.d \leftarrow u.d + w(u, v)$
13. \hspace{2em} $v.\pi \leftarrow u$
14. \hspace{1em} DECREASE-KEY($Q$, $v$, $u.d + w(u, v)$)
SSSP: Dijkstra’s Algorithm

Initial State (with initial tentative distances)
SSSP: Dijkstra’s Algorithm

Step 1: add vertex $s$ to SPT
SSSP: Dijkstra’s Algorithm

Step 1’: update neighbors of $s$
SSSP: Dijkstra’s Algorithm

Step 2: add vertex $y$ through edge $(s, y)$
SSSP: Dijkstra’s Algorithm

Step 2’: update neighbors of y
SSSP: Dijkstra’s Algorithm

Step 3: add vertex z through edge (y, z)
SSSP: Dijkstra’s Algorithm

Step 3’: update neighbors of z
SSSP: Dijkstra’s Algorithm

Step 4: add vertex \( t \) through edge \((y, t)\)
SSSP: Dijkstra’s Algorithm

Step 4’: update neighbors of $t$
SSSP: Dijkstra’s Algorithm

Step 5: add vertex $x$ through edge $(t, x)$
SSSP: Dijkstra’s Algorithm

Step 5’: update neighbors of $x$
SSSP: Dijkstra’s Algorithm

Done
One undirected edge $\Rightarrow$ Two directed edges
SSSP: Dijkstra’s Algorithm

Initial State (with initial tentative distances)
SSSP: Dijkstra’s Algorithm

Step 1: add vertex a to SPT
SSSP: Dijkstra’s Algorithm

Step 1’: update neighbors of $a$
SSSP: Dijkstra’s Algorithm

Step 2: add vertex $b$ through edge $(a, b)$
SSSP: Dijkstra’s Algorithm

Step 2’: update neighbors of \( b \)
SSSP: Dijkstra’s Algorithm

Step 3: add vertex $h$ through edge $(a, h)$
SSSP: Dijkstra’s Algorithm

Step 3’: update neighbors of $h$
SSSP: Dijkstra’s Algorithm

Step 4: add vertex $g$ through edge $(h,g)$
**SSSP: Dijkstra’s Algorithm**

**Step 4’: update neighbors of** $g$
SSSP: Dijkstra’s Algorithm

Step 5: add vertex $f$ through edge $(g, f)$
SSSP: Dijkstra’s Algorithm

Step 5’: update neighbors of $f$
SSSP: Dijkstra’s Algorithm

Step 6: add vertex $c$ through edge $(b, c)$
SSSP: Dijkstra’s Algorithm

Step 6’: update neighbors of c
SSSP: Dijkstra’s Algorithm

Step 7: add vertex $i$ through edge $(c, i)$
SSSP: Dijkstra’s Algorithm

Step 7’: update neighbors of $i$
SSSP: Dijkstra’s Algorithm

Step 8: add vertex $d$ through edge $(c, d)$
SSSP: Dijkstra’s Algorithm

Step 8’: update neighbors of $d$
SSSP: Dijkstra’s Algorithm

Step 9: add vertex e through edge \((f, e)\)
SSSP: Dijkstra’s Algorithm

Step 9’: update neighbors of e
SSSP: Dijkstra’s Algorithm

Done
Dijkstra’s SSSP Algorithm with a Min-Heap
(SSSP: Single-Source Shortest Paths)

**Input:** Weighted graph $G = (V, E)$ with vertex set $V$ and edge set $E$, a weight function $w$, and a source vertex $s \in G[V]$.

**Output:** For all $v \in G[V]$, $v.d$ is set to the shortest distance from $s$ to $v$.

Let $n = |G[V]|$ and $m = |G[E]|$

# INSERTS = $n$

# EXTRACT-MINS = $n$

# DECREASE-KEYS $\leq m$

Total cost

\[ \leq n(\text{cost}_{\text{Insert}} + \text{cost}_{\text{Extract-Min}}) + m(\text{cost}_{\text{Decrease-Key}}) \]
Dijkstra's SSSP Algorithm with a Min-Heap

(SSSP: Single-Source Shortest Paths)

**Input:** Weighted graph $G = (V, E)$ with vertex set $V$ and edge set $E$, a weight function $w$, and a source vertex $s \in G[V]$.

**Output:** For all $v \in G[V]$, $v.d$ is set to the shortest distance from $s$ to $v$.

Let $n = |G[V]|$ and $m = |G[E]|$

For Binary Heap (worst-case costs):

\[
\begin{align*}
&\text{cost}_{\text{Insert}} = O(\log n) \\
&\text{cost}_{\text{Extract-Min}} = O(\log n) \\
&\text{cost}_{\text{Decrease-Key}} = O(\log n)
\end{align*}
\]

\[
\therefore \text{Total cost (worst-case)} = O((m + n) \log n)
\]
Dijkstra’s SSSP Algorithm with a Min-Heap
( SSSP: Single-Source Shortest Paths )

**Input:** Weighted graph \( G = (V, E) \) with vertex set \( V \) and edge set \( E \), a weight function \( w \), and a source vertex \( s \in G[V] \).

**Output:** For all \( v \in G[V] \), \( v.d \) is set to the shortest distance from \( s \) to \( v \).

---

**Dijkstra-SSSP**

1. **for** each vertex \( v \in G.V \) **do**
   2. \( v.d \leftarrow \infty \)
   3. \( v.\pi \leftarrow NIL \)
   4. \( s.d \leftarrow 0 \)
   5. Min-Heap \( Q \leftarrow \emptyset \)
   6. **for** each vertex \( v \in G.V \) **do**
      7. \( INSERT( Q, v ) \)
      8. **while** \( Q \neq \emptyset \) **do**
         9. \( u \leftarrow EXTRACT-MIN( Q ) \)
         10. **for** each \((u, v) \in G.E \) **do**
              11. **if** \( u.d + w(u, v) < v.d \) **then**
                 12. \( v.d \leftarrow u.d + w(u, v) \)
                 13. \( v.\pi \leftarrow u \)
                 14. \( DECREASE-KEY( Q, v, u.d + w(u, v) ) \)

---

Let \( n = |G[V]| \) and \( m = |G[E]| \).

**For Fibonacci Heap (amortized):**

- \( cost_{Insert} = O(1) \)
- \( cost_{Extract-Min} = O(\log n) \)
- \( cost_{Decrease-Key} = O(1) \)

\[ \therefore \text{Total cost (amortized)} = O(m + n \log n) \]
Dijkstra’s SSSP Algorithm with a Min-Heap

(SSSP: Single-Source Shortest Paths)

Input: Weighted graph $G = (V, E)$ with vertex set $V$ and edge set $E$, a weight function $w$, and a source vertex $s \in G[V]$.

Output: For all $v \in G[V]$, $v.d$ is set to the shortest distance from $s$ to $v$.

Let $n = |G[V]|$ and $m = |G[E]|$

# INSERTS = n
# EXTRACT-MINS = n
# DECREASE-KEYS ≤ m

Total cost

\[ \leq n(cost_{Insert} + cost_{Extract-Min}) + m(cost_{Decrease-Key}) \]
Dijkstra’s SSSP Algorithm with a Min-Heap

(SSSP: Single-Source Shortest Paths)

**Input:** Weighted graph $G = (V, E)$ with vertex set $V$ and edge set $E$, a weight function $w$, and a source vertex $s \in G[V]$.

**Output:** For all $v \in G[V]$, $v.d$ is set to the shortest distance from $s$ to $v$.

---

Let $n = |G[V]|$ and $m = |G[E]|$

For Binary Heap (worst-case costs):

- $\text{cost}_{\text{Insert}} = O(\log n)$
- $\text{cost}_{\text{Extract-Min}} = O(\log n)$
- $\text{cost}_{\text{Decrease-Key}} = O(\log n)$

$\therefore$ Total cost (worst-case) 

$= O((m + n) \log n)$
Dijkstra’s SSSP Algorithm with a Min-Heap
 (SSSP: Single-Source Shortest Paths)

Input: Weighted graph $G = (V, E)$ with vertex set $V$ and edge set $E$, a weight function $w$, and a source vertex $s \in G[V]$.

Output: For all $v \in G[V]$, $v \cdot d$ is set to the shortest distance from $s$ to $v$.

```
Dijkstra-SSSP ( G = (V,E), w, s )
1. for each vertex $v \in G.V$ do
2. \hspace{1em} $v \cdot d \leftarrow \infty$
3. \hspace{1em} $v \cdot \pi \leftarrow \text{NIL}$
4. \hspace{1em} $s \cdot d \leftarrow 0$
5. Min-Heap Q \leftarrow \emptyset
6. for each vertex $v \in G.V$ do
7. \hspace{1em} INSERT( Q, v )
8. \hspace{1em} while Q \neq \emptyset do
9. \hspace{2em} $u \leftarrow \text{EXTRACT-MIN}( Q )$
10. \hspace{2em} for each $(u,v) \in G.E$ do
11. \hspace{3em} if $u \cdot d + w(u,v) < v \cdot d$ then
12. \hspace{4em} $v \cdot d \leftarrow u \cdot d + w(u,v)$
13. \hspace{4em} $v \cdot \pi \leftarrow u$
14. \hspace{4em} \text{DECREASE-KEY}( Q, v, u \cdot d + w(u,v) )
```

Let $n = |G[V]|$ and $m = |G[E]|$

For Fibonacci Heap (amortized):

- $\text{cost}_{\text{Insert}} = O(1)$
- $\text{cost}_{\text{Extract-Min}} = O(\log n)$
- $\text{cost}_{\text{Decrease-Key}} = O(1)$

\[ \therefore \text{Total cost (worst-case)} \]
\[ = O(m + n \log n) \]
**Flow Networks**

A **flow network** \( G = (V, E) \) is a directed graph in which each edge \((u, v) \in E\) has a nonnegative **capacity** \(c(u, v)\). Also, if \((u, v) \in E\) then \((v, u) \notin E\). If \((u, v) \notin E\), then we define \(c(u, v) = 0\) for convenience, and we disallow self-loops.

We distinguish two vertices in a flow network: a **source** \(s\) and a **sink** \(t\). For convenience, we assume that each vertex lies on some path from \(s\) to \(t\). The graph is therefore connected and, since each vertex other than \(s\) has at least one entering edge, \(|E| \geq |V| - 1\).
Flows and the Maximum Flow Problem

Let $G = (V, E)$ be a flow network with a capacity function $c$. Let $s$ be the source of the network and let $t$ be the sink.

A **flow** in $G$ is a real-valued function $f: V \times V \to \mathbb{R}$ that satisfies the following two properties:

**Capacity constraint:** For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.

**Flow conservation:** For all $u \in V \setminus \{s, t\}$, we require $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

When $(u, v) \notin E$, there can be no flow from $u$ to $v$, and $f(u, v) = 0$.

The **value** $|f|$ of a flow $f$ is defined as: $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$.

In the **maximum-flow problem**, we are given a flow network $G$ with source $s$ and sink $t$, and we wish to find a flow of maximum value.
Maximum Flow: The Ford-Fulkerson Method

**Input:** A flow network \( G = (V, E) \) with a capacity function \( c \), a source vertex \( s \) and a sink vertex \( t \).

**Output:** A maximum \( s \) to \( t \) flow.

\[
\text{FORD-FULKERSON} \ (G = (V, E), \ s, \ t )
\]

1. for each edge \((u, v) \in G.E\) do
2. \((u, v).f \leftarrow 0\)
3. while there exists a path \( p \) from \( s \) to \( t \) in the residual network \( G_f \) do
4. \( c_f(p) \leftarrow \min\{c_f(u, v) \mid (u, v) \text{ is in } p\}\)
5. for each edge \((u, v) \text{ in } p\) do
6. if \((u, v) \in G.E\) then
7. \((u, v).f \leftarrow (u, v).f + c_f(p)\)
8. else \((v, u).f \leftarrow (v, u).f - c_f(p)\)
Maximum Flow: The Ford-Fulkerson Method

Original Network

Original Network
Step 1: Augmenting Path

Residual capacity = 4
Step 1: Augmenting Path

Residual capacity = 4

Increase flow by 4 along path
s → v₁ → v₃ → v₂ → v₄ → t
Step 1: Augmenting Path

Step 1: Updated Flow

Current $s$ to $t$ flow = 4
Maximum Flow: The Ford-Fulkerson Method

Step 2: Residual Network

Step 1: Updated Flow
Maximum Flow: The Ford-Fulkerson Method

Step 2: Augmenting Path

Step 1: Updated Flow

Residual capacity = 4
Maximum Flow: The Ford-Fulkerson Method

Step 2: Augmenting Path

Step 2: Updating Flow

Residual capacity = 4

Increase flow by 4 along path
\[ s \rightarrow v_4 \rightarrow v_1 \rightarrow v_3 \rightarrow t \]
Maximum Flow: The Ford-Fulkerson Method

Step 2: Augmenting Path

Step 2: Updated Flow

Current s to t flow = 8
Maximum Flow: The Ford-Fulkerson Method

Step 3: Residual Network

Step 2: Updated Flow
Maximum Flow: The Ford-Fulkerson Method

Step 3: Augmenting Path

Step 2: Updated Flow

Residual capacity = 4
Maximum Flow: The Ford-Fulkerson Method

Step 3: Augmenting Path

Residual capacity = 4

Step 3: Updating Flow

Increase flow by 4 along path $s \rightarrow v_1 \rightarrow v_4 \rightarrow v_3 \rightarrow t$
Maximum Flow: The Ford-Fulkerson Method

Step 3: Augmenting Path

Step 3: Updated Flow

Current $s$ to $t$ flow = 12
Maximum Flow: The Ford-Fulkerson Method

Step 4: Residual Network

Step 3: Updated Flow
Maximum Flow: The Ford-Fulkerson Method

Step 4: Augmenting Path

Step 3: Updated Flow

Residual capacity = 7
Maximum Flow: The Ford-Fulkerson Method

Step 4: Augmenting Path

Step 4: Updating Flow

Increase flow by 7 along path
\[ s \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow t \]
Maximum Flow: The Ford-Fulkerson Method

Step 4: Augmenting Path

Step 4: Updated Flow

Current $s$ to $t$ flow = 19
Maximum Flow: The Ford-Fulkerson Method

Step 5: Residual Network

Step 4: Updated Flow
Maximum Flow: The Ford-Fulkerson Method

Step 5: Augmenting Path

Step 4: Updated Flow

Residual capacity = 4
**Maximum Flow: The Ford-Fulkerson Method**

**Step 5: Augmenting Path**

![Graph showing the augmentation step](image)

Residual capacity = 4

**Step 5: Updating Flow**

![Graph showing the update step](image)

Increase flow by 4 along path $s \rightarrow v_1 \rightarrow v_3 \rightarrow t$
Maximum Flow: The Ford-Fulkerson Method

Step 5: Augmenting Path

Step 5: Updated Flow

Current $s$ to $t$ flow = 23
Maximum Flow: The Ford-Fulkerson Method

Step 6: Residual Network

Step 5: Updated Flow

No augmenting path!
Maximum Flow: The Ford-Fulkerson Method

Step 6: Residual Network

Step 6: No Update

No augmenting path!

Done!
Maximum $s$ to $t$ flow = 23
The Ford-Fulkerson Method: Running Time

The running time of *Ford-Fulkerson* depends on how we find the augmenting paths. If we choose them poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value (e.g., might happen when the capacities are irrational numbers).

In practice, the capacities are often integral. If the capacities are rational numbers, we can apply an appropriate scaling transformation to make them all integral. If $f^*$ denotes a maximum flow in the transformed network, then a straightforward implementation of *Ford-Fulkerson* requires to find an augmenting path at most $|f^*|$ times, since each augmentation increases the flow value by at least one unit.
Once the residual network $G_f$ is known, an augmenting path can be found in $O(m + n)$ time using either a depth-first or a breadth-first search, where $n = |V|$ and $m = |E|$. It is also easy to maintain the network, capacities and flows in a way that allows one to find $G_f$ and update the flows in $O(m + n)$ time during each augmentation.

The running time of $\text{FORD-FULKERSON}$ is thus $O((m + n)|f^*|)$ which is simply $O(m|f^*|)$ as $m = \Omega(n)$. 

\[
\text{FORD-FULKERSON (} G = (V,E), \ s, \ t \text{)}
\]

1. for each edge $(u,v) \in G.E$ do
2. \hspace{1em} $(u,v).f \leftarrow 0$
3. while there exists a path $p$ from $s$ to $t$ in the residual network $G_f$ do
4. \hspace{1em} $c_f(p) \leftarrow \min\{c_f(u,v) | (u,v) \text{ is in } p\}$
5. \hspace{1em} for each edge $(u,v)$ in $p$ do
6. \hspace{2em} if $(u,v) \in G.E$ then
7. \hspace{3em} $(u,v).f \leftarrow (u,v).f + c_f(p)$
8. \hspace{2em} else $(v,u).f \leftarrow (v,u).f - c_f(p)$
The Edmonds-Karp Algorithm

\begin{algorithm}
\textsc{Ford-Fulkerson} ( \( G = (V,E) \), s, t )
\begin{enumerate}
\item for each edge \( (u,v) \in G.E \) do
\item \( (u,v).f \leftarrow 0 \)
\item while there exists a path \( p \) from \( s \) to \( t \) in the residual network \( G_f \) do
\item \( c_f(p) \leftarrow \min\{c_f(u,v) | (u,v) \text{ is in } p\} \)
\item for each edge \( (u,v) \) in \( p \) do
\item \textbf{if} \( (u,v) \in G.E \) \textbf{then}
\item \( (u,v).f \leftarrow (u,v).f + c_f(p) \)
\item \textbf{else} \( (v,u).f \leftarrow (v,u).f - c_f(p) \)
\end{enumerate}
\end{algorithm}

The Edmonds-Karp algorithm is an implementation of the \textsc{Ford-Fulkerson} method in which the augmenting path \( p \) in line 3 is found using a breadth-first search. That is, \( p \) is chosen as a shortest path from \( s \) to \( t \) in the residual network, where each edge has unit distance (weight).

One can show that the Edmonds-Karp algorithm runs in \( O(m^2n) \) time, where \( n = |G.V| \) and \( m = |G.E| \).
**The Edmonds-Karp Algorithm**

**Lemma 26.7 (CLRS):** If the Edmonds-Karp algorithm is run on a flow network \( G = (V, E) \) with source \( s \) and sink \( t \), then for all vertices \( v \in G. V \setminus \{s, t\} \), the shortest path distance \( \delta_f(s, v) \) in the residual network \( G_f \) increases monotonically with each flow augmentation.

**Proof:** Let’s assume for contradiction that for some vertex \( v \in G. V \setminus \{s, t\} \), there is a flow augmentation that causes the shortest-path distance from \( s \) to \( v \) to decrease.

Let \( f \) be the flow just before the first augmentation that decreases some shortest-path distance, and let \( f' \) be the flow just afterward.

Let \( v \) be the vertex with the minimum \( \delta_{f'}(s, v) \) whose distance was decreased by the augmentation, so that \( \delta_{f'}(s, v) < \delta_f(s, v) \).
The Edmonds-Karp Algorithm

Proof (Continued): Let $p = s \rightsquigarrow u \rightarrow v$ be a shortest path from $s$ to $v$ in $G_f$, so that $(u, v) \in E_f$, and $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$.

Because of the way we chose $v$, we know that $\delta_{f'}(s, u) \geq \delta_f(s, u)$. Then we must have $(u, v) \notin E_f$, as otherwise by triangle inequality:

$$\delta_f(s, v) \leq \delta_f(s, u) + 1 \leq \delta_{f'}(s, u) + 1 = \delta_{f'}(s, v),$$

which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$.

How can we have $(u, v) \notin E_f$ and $(u, v) \in E_{f'}$? The augmentation must have increased the flow from $v$ to $u$. The Edmonds-Karp algorithm always augments flow along shortest paths, and therefore the shortest path from $s$ to $u$ in $G_f$ has $(v, u)$ as its last edge.

Therefore, $\delta_f(s, v) = \delta_f(s, u) - 1 \leq \delta_{f'}(s, u) - 1 = \delta_{f'}(s, v) - 2$, which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$. 
The Edmonds-Karp Algorithm

**Theorem 26.8 (CLRS):** If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source $s$ and sink $t$, then the total number of flow augmentations performed by the algorithm is $O(mn)$, where $n = |G.V|$ and $m = |G.E|$.

**Proof:** We say that an edge $(u, v)$ in a residual network $G_f$ is **critical** on an augmenting path $p$ if $c_f(p) = c_f(u, v)$.

After we have augmented flow along an augmenting path, any critical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical.

We will show that each of the $m$ edges can become critical at most $n/2$ times.
**The Edmonds-Karp Algorithm**

**Proof (continued):** Let $u, v \in G$. $V$ be connected by $(u, v) \in G. E$.

Since augmenting paths are shortest paths, when $(u, v)$ is critical for the first time, we have $\delta_f(s, v) = \delta_f(s, u) + 1$.

Once the flow is augmented, the edge $(u, v)$ disappears from the residual network. It cannot reappear later on another augmenting path until after the flow from $u$ to $v$ is decreased, which occurs only if $(v, u)$ appears on an augmenting path. If $f'$ is the flow in $G$ when this event occurs, then we have $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$.

Since $\delta_f(s, v) \leq \delta_{f'}(s, v)$ by Lemma 26.7, we have:

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2.$$
The Edmonds-Karp Algorithm

Proof (Continued): Consequently, from the time \((u, v)\) becomes critical to the time when it next becomes critical, the distance of \(u\) from \(s\) increases by at least 2. The distance of \(u\) from \(s\) is initially at least 0. The intermediate vertices on a shortest path from \(s\) to \(u\) cannot contain \(s\), \(u\), or \(t\) (since \((u, v)\) on an augmenting path implies that \(u \neq t\)). Therefore, until \(u\) becomes unreachable from \(s\), if ever, its distance is at most \(n - 2\). Thus, after the first time that \((u, v)\) becomes critical, it can become critical at most \((n - 2)/2 = n/2 - 1\) times more, for a total of at most \(n/2\) times.

Since there are \(O(m)\) pairs of vertices that can have an edge between them in a residual network, the total number of critical edges during the entire execution of the algorithm is \(O(mn)\). Each augmenting path has at least one critical edge, and hence the theorem follows.
A cut \((S, T)\) of flow network \(G = (V, E)\) is a partition of \(V\) into \(S\) and \(T = V \setminus S\) such that \(s \in S\) and \(t \in T\).

(Unlike the “cut” used for MST’s, here the graph is directed, and we insist that \(s \in S\) and \(t \in T\).)

If \(f\) is a flow, then the net flow \(f(S, T)\) across \((S, T)\) is defined to be

\[
f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u).
\]

The capacity of the cut \((S, T)\) is:

\[
c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v).
\]

A minimum cut of a network is a cut of minimum capacity.
LEMMA 26.4 (CLRS): Let $f$ be a flow in a flow network $G$ with source $s$ and sink $t$, and let $(S, T)$ be any cut of $G$. Then the net flow across $(S, T)$ is $f(S, T) = |f|$.

COROLLARY 26.5 (CLRS): The value of any flow $f$ in a flow network $G$ is bounded from above by the capacity of any cut of $G$.

Corollary 26.5 implies that in a flow network:

\[ \text{value of a maximum flow} \leq \text{capacity of a minimum cut} \]
Cuts of Flow Networks (Max-flow Min-cut Theorem)

Theorem 26.6 below says that, in fact, in a flow network:

\[
\text{value of a maximum flow} = \text{capacity of a minimum cut}
\]

**Theorem 26.6 (CLRS):** If \( f \) is a flow in a flow network \( G = (V, E) \) with source \( s \) and sink \( t \), then the following conditions are equivalent:

1. \( f \) is a maximum flow in \( G \).
2. The residual network \( G_f \) contains no augmenting paths.
3. \( |f| = c(S, T) \) for some cut \( (S, T) \) of \( G \).
The Maximum Matching Problem

Given an undirected graph $G = (V, E)$, a **matching** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of $M$ is incident on $v$. We say that a vertex $v \in V$ is matched by the matching $M$ if some edge in $M$ is incident on $v$; otherwise, $v$ is unmatched.

A **maximum matching** is a matching of maximum cardinality, that is, a matching $M$ such that for any matching $M'$, we have $|M| \geq |M'|$. 
The Maximum Bipartite Matching Problem

We shall restrict our attention to finding maximum matchings in bipartite graphs: graphs in which the vertex set can be partitioned into $V = L \cup R$, where $L$ and $R$ are disjoint and all edges in $E$ go between $L$ and $R$. We further assume that every vertex in $V$ has at least one incident edge.

A bipartite graph
The Maximum Bipartite Matching Problem

We shall restrict our attention to finding maximum matchings in bipartite graphs: graphs in which the vertex set can be partitioned into \( V = L \cup R \), where \( L \) and \( R \) are disjoint and all edges in \( E \) go between \( L \) and \( R \). We further assume that every vertex in \( V \) has at least one incident edge.

A bipartite matching
The Maximum Bipartite Matching Problem

We shall restrict our attention to finding maximum matchings in bipartite graphs: graphs in which the vertex set can be partitioned into $V = L \cup R$, where $L$ and $R$ are disjoint and all edges in $E$ go between $L$ and $R$. We further assume that every vertex in $V$ has at least one incident edge.

A maximum bipartite matching

$L$  $R$
We shall restrict our attention to finding maximum matchings in bipartite graphs: graphs in which the vertex set can be partitioned into $V = L \cup R$, where $L$ and $R$ are disjoint and all edges in $E$ go between $L$ and $R$. We further assume that every vertex in $V$ has at least one incident edge.
Maximum Bipartite Matching using Network Flow

Given a bipartite graph $G = (V, E)$ with $V = L \cup R$, where $L$ and $R$ are disjoint and all edges in $E$ go between $L$ and $R$. 
First, direct all edges from $L$ to $R$. 

![Diagram]

$L$ \hspace{2cm} $R$
Then add a source $s$ and a sink $t$.

For every vertex $v \in L$, add an edge $(s, v)$ directed from $s$ to $v$.

For every vertex $v \in R$, add an edge $(v, t)$ directed from $v$ to $t$. 
Maximum Bipartite Matching using Network Flow

For every edge \((u, v)\) in this new directed graph, set capacity \(c(u, v) = 1\).
Maximum Bipartite Matching using Network Flow

Now, find a maximum $s$ to $t$ flow $f^*$ in this new graph using **Ford-Fulkerson**.

One can show that $|f^*|$ will always be an integer and will be equal to the maximum matching in the original bipartite graph.

Since $|f^*| < n = |L \cup R|$, running time will be $O(mn)$, where $m = |E|$.