A graph $G$ consists of a pair $(V; E)$, where $V$ is the set of vertices and $E$ the set of edges. We write $V(G)$ for the vertices of $G$ and $E(G)$ for the edges of $G$.

If no two edges have the same endpoints we say there are no multiple edges, and if no edge has a single vertex as both endpoints we say there are no loops. A graph with no loops and no multiple edges is a simple graph. A complete graph is a graph, where any two vertices are adjacent.

This graph is a connected graph, if each pair of vertices $v, w$ is connected by a sequence of vertices and edges. Each of the following graphs are connected, but if we consider them as a single graph then it is not connected.

A graph $G = (V,E)$ that is not simple can be represented by using multisets: a loop is a multiset $\{v,v\} = \{2 \cdot v\}$ and multiple edges are represented by making $E$ a multiset. We will rarely need to use this level of detail in our discussions. A general graph that is not connected, has loops, and has multiple edges is shown here:
A complete graph is a simple graph in which any two vertices are adjacent.
A graph $G$ is **bipartite** if its vertices can be partitioned into two parts, $\{x_1, x_2, \ldots, x_n\}$ and $\{y_1, y_2, \ldots, y_m\}$, so that all edges join some $x_i$ to some $y_j$; no two vertices $x_i$ and $x_j$ are adjacent, nor are any vertices $x_i$ and $y_j$.

A **star** is a complete bipartite graph $G [X, Y]$ with $|X| = 1$ or $|Y| = 1$.

A **complete bipartite** graph $K_{m,n}$ is a graph that has its vertex set partitioned into two subsets of $m$ and $n$ vertices, respectively with an edge between two vertices if and only if one vertex is in the first subset and the other vertex is in the second subset. A **star** is a $K_{1,n}$.

A path is a **simple** graph whose vertices can be arranged in a linear sequence in such a way that two vertices are adjacent if they are consecutive in the sequence, and are nonadjacent otherwise.

A **cycle** on three or more vertices is a simple graph whose vertices can be arranged in a cyclic sequence in such a way that two vertices are adjacent if they are consecutive in the sequence, and are nonadjacent otherwise; a cycle on one vertex consists of a single vertex with a loop, and a cycle on two vertices consists of two vertices joined by a pair of parallel edges.

The **length** of a path or a cycle is the number of its edges. A path or cycle of length $k$, is called a $k$-path or $k$-cycle, respectively. A 3-cycle is often called a triangle, a 4-cycle, a quadrilateral, a 5-cycle a pentagon, a 6-cycle a hexagon, and so on.
A connected graph without cycles is defined as a **tree**. A graph without cycles is called an **acyclic graph** or a **forest**. So each component of a forest is a tree. A forest may consist of just a single tree.

**Theorem.** A simple graph is a tree if and only if any two distinct vertices are connected by a unique path.

A graph which can be drawn in the plane in such a way that edges meet only at points corresponding to their common ends is called a **planar graph**.

If G is a connected graph on n vertices, a **spanning tree** for G is a subgraph of G that is a tree on all n vertices. Note that Every connected graph has a spanning tree.

**Theorem.** If G is connected, it has at least n - 1 edges; moreover, it has exactly n - 1 edges, if and only if it is a tree. Also, G is a tree if and only if there is a unique path between any two vertices.

A **directed graph** (or **digraph**) is a graph, where the edges have a direction associated with them. An arrow \((x, y)\) is considered to be directed from \(x\) to \(y\); \(y\) is called the **head** and \(x\) is called the **tail** of the arrow; \(y\) is said to be a **direct successor** of \(x\) and \(x\) is said to be a **direct predecessor** of \(y\). For a vertex, the number of head ends adjacent to a vertex is called the **in-degree** of the vertex and the number of tail ends adjacent to a vertex is its **out-degree**

Often in operation research, a directed graph is called a network, the vertices are called nodes and the edges are called arcs.

A **network** is a digraph with a designated **source** \(s\) and **sink** or **target** \(t\). In addition, each arc \(e\) has a **positive capacity**, \(c(e)\). A network can be used to model traffic in a road system, circulation with demands, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes.
Many of the concepts and terminology for graphs are also valid for digraphs. However, there are many concepts of digraphs involving the notion of orientation that apply only to digraphs.

**Incidence and Adjacency Matrices**

Although drawings are a convenient means of specifying graphs, they are clearly not suitable for storing graphs in computers, or for applying mathematical methods to study their properties. For these purposes, we consider two matrices associated with a graph.

**Incidence Matrix**: Let $G = (V,E)$ be an undirected graph. The incidence matrix with respect to this ordering of $V$ and $E$ is the $n \times m$ matrix $M_G := (m_{ve})$, where $m_{ve}$ is the number of times (0, 1, or 2) that vertex $V$ and edge $E$ are incident.

**Adjacency Matrix**: Let $G = (V,E)$ be a directed or undirected graph. The adjacency matrix of $G$ is the $n \times n$ matrix $A_G := (a_{uv})$, where $a_{uv}$ is the number of edges joining vertices $u$ and $v$ each loop counting as two edges.

**Isomorphism of Graphs**

The simple graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if there exists a one-to-one and onto function $f$ from $V_1$ to $V_2$ with the property that $a$ and $b$ are adjacent in $G_1$ if and only if $f(a)$ and $f(b)$ are adjacent in $G_2$, for all $a$ and $b$ in $V_1$. Such a function $f$ is called an **isomorphism**. Two simple graphs that are not isomorphic are called **non-isomorphic**.
Isomorphic graphs

To show that two graphs $G$ and $H$ are isomorphic, we only need to show that by permuting some rows of the adjacency matrix $A_G$, we obtain the adjacency matrix $A_H$.

Non-isomorphic graphs

Euler Paths and Cycles

The first problem in graph theory dates to 1735, and is called the Seven Bridges of Königsberg. In Königsberg, were two islands, connected to each other and the mainland by seven bridges, as shown here:

The question, which made its way to Euler, was whether it was possible to take a walk and cross over each bridge exactly once; Euler showed that it is not possible. We can represent this problem as a graph, as follows:
The two sides of the river are represented by the top and bottom vertices, and the islands by the middle two vertices. There are two possible interpretations of the question, depending on whether the goal is to end the walk at its starting point.

A walk in a graph is a sequence of vertices and edges,

\[ v_1, e_1, v_2, e_2, \ldots, v_k, e_k, v_{k+1} \]

such that the endpoints of edge \( e_i \) are \( v_i \) and \( v_{i+1} \). In general, the edges and vertices may appear in the sequence more than once. If \( v_1 = v_{k+1} \), the walk is a closed walk. We will deal first with the case in which the walk is to start and end at the same place.

A successful walk in Königsberg corresponds to a closed walk in the graph in which every edge is used exactly once.

**Euler’s Theorem.**

1) A connected graph has an Euler cycle, if and only if every vertex has an even degree.

2) A connected graph has an open Euler path, if and only if it has exactly two odd vertices.

3) A connected digraph has an Euler cycle, if and only if the indegree and outdegree of every vertex are equal.

4) A connected graph has an open Euler path, if and only if the indegree and outdegree of all vertices are equal, except two. For one vertex, the indegree exceeds the outdegree by one; and for the other, the outdegree exceeds the indegree by one.

**Exercise:** 1) Can someone cross all the bridges shown in this map exactly once and return to the starting point?

2) Determine whether each of the directed graphs shown has an Euler circuit. Construct an Euler circuit if one exists.

**Hamilton Paths and Circuits**

A simple path in a graph \( G \) that passes through every vertex exactly once is called a Hamilton path, and a simple circuit in a graph \( G \) that passes through every vertex exactly once is called a Hamilton circuit. That is, the simple path \( x_0, x_1, \ldots, x_{n-1}, x_n \) in the graph \( G = (V, E) \) is a Hamilton path if \( V = \{ x_0, x_1, \ldots, x_{n-1}, x_n \} \) and \( x_i = x_j \) for \( 0 \leq i < j \leq n \), and the simple circuit \( x_0, x_1, \ldots, x_{n-1}, x_n, x_0 \) (with \( n > 0 \)) is a Hamilton circuit if \( x_0, x_1, \ldots, x_{n-1}, x_n \) is a Hamilton path.
Dirac’s Theorem. If G is a simple graph with \( n \geq 3 \) vertices such that the degree of every vertex in G is at least \( n/2 \), then G has a Hamilton circuit.

Ore’s Theorem. If G is a simple graph with \( n \geq 3 \) vertices such that \( \text{deg}(u) + \text{deg}(v) \geq n \) for every pair of nonadjacent vertices \( u \) and \( v \) in G, then G has a Hamilton circuit.

Exercise: Show that the Petersen graph, shown here, does not have a Hamilton circuit, but that the subgraph obtained by deleting a vertex \( v \), and all edges incident with \( v \), does have a Hamilton circuit.

Minimum Cost Spanning Tree
A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected. By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

• A connected graph G can have more than one spanning tree.
• All possible spanning trees of graph G, have the same number of edges and vertices.
• The spanning tree does not have any cycle (loops).
• Removing one edge from the spanning tree will make the graph disconnected,
• Adding one edge to the spanning tree will create a circuit or loop.

Prim’s Minimum Spanning Tree Algorithm
1) Initialize a tree with an arbitrary vertex from the graph.
2) Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3) Repeat step 2 (until all vertices are in the tree).
Consider the following example:

![Graph Image]

**Solution:**
1) Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others. We remove the edge 9: A → B and 1: C → C.
2) The fact that in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree. Thus, we may start our three with any arbitrary vertex; we choose S.
3) The vertices A and C are the only vertices connected to S. We choose the edge 7: S → A, as it is lesser than the other.
4) Next, we choose a vertex with minimum weight connected to S or A. The edge 3: A → C is less than 8: S → C and 6: A → B; so we must include the vertex C in our spanning tree.
5) B is connected to A and C with 6: A → B, and 4: C → B. The vertex D is connected to C with 3: C → D. Clearly D must be chosen.
6) From the fact that D is part of the vertices in the spanning tree and the edges 2: D → B and 2: D → T are smaller than any other remaining edges, we conclude that these two edges must be chosen. Hence we have:

![Kruskal's Minimum Spanning Tree Diagram]

**Kruskal's Minimum Spanning Tree Algorithm**

We explain the algorithm using the example of the Prim's algorithm.

1) We start with the first step of Prim’s algorithm.
2) The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).
3) We construct a tree by adding edges to the graph beginning from the one which has the least weight. So we use 2: B → D, 2: D → T, 3: A → D and 4L C → B. The edges 5: B → T and 6: A → B are not included, since they would produce cycles. The last edge will be 7: S → A.

**Borůvka's Minimum Spanning Tree Algorithm**

This algorithm which finds a minimum spanning tree in a graph for which all edge weights are distinct is the oldest minimum spanning tree algorithm. It was discovered by Borůvka in 1926, long before computers even existed. The algorithm was published as a method of constructing an efficient electricity network.

1) Initialize all vertices as individual components (or sets).
2) Initialize MST (minimum spanning tree set) as empty.
3) While there are more than one components, do following for each component.
   a) Find the closest weight edge that connects this component to any other component.
   b) Add this closest edge to MST if not already added.
5) Return to MST.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Components</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Graph" /></td>
<td>{A}</td>
<td>This is our original weighted graph. The numbers near the edges indicate their weight. Initially, every vertex by itself is a component (blue circles).</td>
</tr>
<tr>
<td><img src="image2.png" alt="Graph" /></td>
<td>{A,B,D,F} {C,E,G}</td>
<td>In the first iteration of the outer loop, the minimum weight edge out of every component is added. Some edges are selected twice (AD, CE). Two components remain.</td>
</tr>
<tr>
<td><img src="image3.png" alt="Graph" /></td>
<td>{A,B,C,D,E,F,G}</td>
<td>In the second and final iteration, the minimum weight edge out of each of the two remaining components is added. These happen to be the same edge. One component remains and we are done. The edge BD is not considered because both endpoints are in the same component.</td>
</tr>
</tbody>
</table>

**Shortest-Path Problem**

The problem of finding the shortest path connecting two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
Dijkstra's algorithm for the shortest path
Let us understand with the following example:

**Initial**
Select the root node to form the green set S1. Assign the path length 0 to this node. Put all other nodes in the pink set S2.

**Selection**
Compute the lengths of the paths to all nodes directly reachable from S1 through a node in S1. Select the node in S2 with the smallest path length.
Let the edge connecting this node with S1 be (i, j). Add this edge to the shortest path tree. Add node j to the set S1 and delete it from the set S2.

**Finish**
If the set S1 includes all the nodes (all the vertices are green), stop with the shortest path tree. Otherwise repeat the Selection step.
Here are all the steps for finding the shortest path for this problem:

**Example 2.** Use Dijkstra’s algorithm to find the length of a shortest path between the vertices a and z in the weighted graph displayed here:

Here is another way of using this algorithm:
1) Create a visited set $S_1$ that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.

3) Set the initial node as visited. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set $S_2$.

4) For the current node, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.

5) When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set $S_2$. A visited node will never be checked again.

6) If the destination node has been marked visited or if the smallest tentative distance among the nodes in the unvisited set is infinity (this occurs in the case of digraph, when there is no connection between the latest visited node and remaining unvisited nodes), then stop. The algorithm has finished.

7) Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 4.

All the steps involved to obtain the shortest path:

An important problem involving weighted graphs is the **traveling problem**. Consider the following problem: A traveling salesperson wants to visit each of $n$ cities exactly once and return to his starting point. In which order should he visit these cities to travel the minimum total distance? Note that in this shortest path problem, the starting point is also the end point. In the case of non-weighted graph, the traveling salesperson problem is finding a Hamiltonian circuit.
Exercise: Find the length of a shortest path between a and z in the given weighted graphs.

Minimum Spanning Tree Versus Shortest Path
The Minimum Spanning Tree and Shortest Path algorithms looks similar; however, they focus on 2 different requirements.
In Minimum Spanning Tree (MST), we must construct a tree that reaches every single vertex and the total cost of reaching each vertex must be minimum among all possible combinations.
In Shortest Path, requirement is to reach destination vertex from source vertex with lowest possible cost (shortest weight). So here we do not worry about reaching each vertex instead only focus on source and destination vertices and that’s where lies the difference.

The Critical Path Method (CPM)

CPM is a mathematically based algorithm for scheduling a set of project activities. It is commonly used with all forms of projects, including construction, software development, research projects, product development, engineering, and plant maintenance, among others. Any project with interdependent activities can apply this method of scheduling.

Given a network of activities, the first problem of interest is to determine the length of time required to complete the project and the set of critical activities that control the project completion time. Suppose that in a given project activity network there are m nodes, n arcs (i.e. activities) and an estimated duration time, $C_{ij}$, associated with each arc (i to j) in the network. The beginning node of an arc corresponds to the start of the associated activity and the end node to the completion of an activity.

Each arc has two roles: it represents an activity and it defines the precedence relationships among the activities. Sometimes it is necessary to add arcs that only represent precedence relationships. These dummy arcs are represented by dashed arrows. In our example, the arc from B to A represents a dummy activity. The purpose of adding dummy arcs is have only one target.
Problem Statement
We introduce a small example that will be used to illustrate various aspects of CPM. Suppose we are constructing a new building; the required construction activities are shown in the table below along with the estimated duration of each activity and any immediate predecessors. An immediate predecessor of an activity $y$ is an activity $x$ that must be completed no later than the starting time of activity $y$. When an activity has more than one immediate predecessor, all of them must be completed before the activity can begin.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Duration (weeks)</th>
<th>Predecessor(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>None</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>8</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>B, E</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>F</td>
</tr>
</tbody>
</table>

There are many questions to be answered when scheduling a complex project but two of the important questions are:

- What is the total time required to complete the project if no delays occur?
- When do the individual activities need to start and to finish to meet the project deadline?
- When can the individual activities start and finish (at the earliest) if no delays occur?
- What are the critical bottleneck activities?

The figure below shows a network representation of the project information from the table above.

- What is the total time required to complete the project?

If we add up the times required for all of the activities, we get 28 weeks. However, this is not the best answer to the question, since some of the activities can be performed at the same time. Instead, to determine the total time required, we want to consider the length of each path through the network. A path through the network is a route made up of nodes and arcs that traverses the network from the start node to the finish node. The length of a path is the sum of the durations of the activities on the nodes along the path. In this simple example, there are two paths through the network:

- start -> A -> B -> F -> G -> finish, with a length of 13
- start -> A -> C -> D -> E -> F -> G -> finish, with a length of 25

The project duration will be no longer than the longest path through the network. Therefore, the total time required to complete the project equals the length of the longest path through the network -- and this longest path is called the critical path. In the example, the total time to complete the project should be 25 weeks.
Here is an example of how to use the Kruskal's minimum spanning tree algorithm to solve a scheduling problem:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Previous</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>A &amp; F</td>
<td>max {2, 5} = 5</td>
</tr>
<tr>
<td>B</td>
<td>A &amp; D</td>
<td>max {5, 5 + 3} = 8</td>
</tr>
<tr>
<td>G</td>
<td>F &amp; D</td>
<td>max {5 + 4, 2 + 4} = 9</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>8 + 4 = 12</td>
</tr>
<tr>
<td>E</td>
<td>B &amp; D &amp; G</td>
<td>max {8 + 6, 5 + 4, 9 + 2} = 14</td>
</tr>
<tr>
<td>H</td>
<td>G &amp; E</td>
<td>max {9 + 5, 14 + 4} = 18</td>
</tr>
<tr>
<td>T</td>
<td>C &amp; E &amp; H</td>
<td>max {12 + 3, 14 + 5, 18 + 5} = 23</td>
</tr>
</tbody>
</table>
Planar Graph

An undirected graph $H$ is called a **minor** of the graph $G$, if $H$ can be formed from $G$ by deleting edges and vertices and by contracting edges. The following diagram illustrates how to obtain the minor $H$ of $G$. First construct a subgraph of $G$ by deleting the dashed edges (and the resulting isolated vertex), and then contract the gray edge (merging the two vertices it connects):

![Diagram](image1.png)

**Klaus Wagner theorem (1937).** A graph is planar if and only if it does not contain a minor isomorphic to $K_5$ (the complete graph on five vertices) or $K_{3,3}$ (complete bipartite graph on six vertices, three of which connect to each of the other three, also known as the utility graph).

![Utility Graph $K_{3,3}$](image2.png)

An animation showing that the Petersen graph contains a minor isomorphic to the $K_{3,3}$ graph
[https://upload.wikimedia.org/wikipedia/commons/0/0d/Kuratowski.gif?1486327215731](https://upload.wikimedia.org/wikipedia/commons/0/0d/Kuratowski.gif?1486327215731)

A planar representation of a graph splits the plane into regions bounded by edges, including the outer, infinitely large region. For example, the planar representation of $Q_3$ splits the plane into six regions.

**Euler’s Formula:** Let $G$ be a connected planar simple graph with $e$ edges and $v$ vertices.
Let $r$ be the number of regions in a planar representation of $G$. Then
\[ r = e - v + 2. \]
The Graph $Q_3$ with twelve edges ($e = 12$) and eight vertices ($v = 8$) splits the plane into six regions. We have $r = 12 - 8 + 2 = 6$.

**Corollary 1.** If $G$ is a connected planar simple graph with $e$ edges and $v$ vertices, where $v \geq 3$, then $e \leq 3v - 6$.

**Corollary 2.** If $G$ is a connected simple planar graph, then $G$ has a vertex of degree less than five.

The complete graph $K_5$ has five vertices and 10 edges. However, the inequality $e \leq 3v - 6$ is not satisfied for this graph because $e = 10$ and $3v - 6 = 9$. Therefore, $K_5$ is not planar.

**Corollary 3.** If a connected planar simple graph has $e$ edges and $v$ vertices with $v \geq 3$ and no circuits of length three, then $e \leq 2v - 4$.

The graph $K_{3,3}$ has nine edges and six vertices. However, the inequality $e \leq 2v - 4$ is not satisfied for this graph because $e = 9$ and $2v - 4 = 8$. Therefore, $K_{3,3}$ is not planar.

**Applications of Planar Graphs**
Planarity of graphs plays an important role in the design of electronic circuits. We can model a circuit with a graph by representing components of the circuit by vertices and connections between them by edges. We can print a circuit on a single board with no connections crossing if the graph representing the circuit is planar. When this graph is not planar, we must turn to more expensive options. For example, we can partition the vertices in the graph representing the circuit into planar subgraphs. We then construct the circuit using multiple layers.

The planarity of graphs is also useful in the design of road networks. Suppose we want to connect a group of cities by roads. We can model a road network connecting these cities using a simple graph with vertices representing the cities and edges representing the highways connecting them. We can build this road network without using underpasses or overpasses if the resulting graph is planar.

**What is Greedy Algorithm?**
There is no formal definition of what exactly a greedy algorithm is. So let’s consider an informal definition of what greedy algorithm usually looks like. A greedy algorithm, as the name suggests, always makes the choice that looks best at that moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

In general, greedy algorithms have five components:

- A candidate set, from which a solution is created.
- A selection function, which chooses the best candidate to be added to the solution.
- A feasibility function, that is used to determine if a candidate can be used to contribute to a solution.
- An objective function, which assigns a value to a solution, or a partial solution.
- A solution function, which will indicate when we have discovered a complete solution.

All the minimum spanning tree and shortest path algorithms described above are greedy algorithms.
Maximal-Flow Problem

A flow network (also known as a transportation network) is a weighted connected digraph $G = (V, E)$ that contains no loops and no multiple arcs together with a non-negative function $c: V \times V \rightarrow \mathbb{R}$, called the capacity function. It is assumed that each arc has a capacity and each arc receives a flow. A flow of a transportation network is a set of non-negative numbers assigned to each arc of the network and called the flow on that node, satisfying the two conditions:

1) The flow along any arc $(u,v)$ cannot exceed $c(u,v)$.

2) At any node except the source and sink, the sum of the flows on the in-coming arcs equals the sum of the flows of the outgoing arcs.

The sum of the flows on the arcs directed out from the source must equal the sum of the flows on the arcs directed into the sink. These sums are called the value of the flow. A flow is said to be maximal, if the value is as large as possible.

Any transportation network has a maximal flow. Usually there are many different flows that are maximal. If the flow equals the capacity on any arc, the arc is said to be saturated.

Consider a path from the source $S$ to the sink $T$ in which an arc with non-zero flow is traversed in the wrong (reverse) direction. This path is called a pseudo-path and any arc traversed backward is called pseudo-arc.

Ford-Fulkerson Algorithm (1956)

We shall use the following transportation problem to explain about Ford-Fulkerson Algorithm for finding the maximal-flow:

Solution:

Numbers in parenthesis are the unused units. From step 1 to step 4, we arbitrary use some paths from the source $S$ to the sink $T$, and ship the maximum number of units possible. At step 4, we realize that we have some units left. Although there are some unsaturated arcs, but the next arc is saturated. At this point we try to use the Ford-Fulkerson algorithm.

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Step 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
</tbody>
</table>

$\min \{5, 3, 4\} = 3$

$\min \{2, 2, 3\} = 2$
### Step 3

![Graph](image)

\[ S \rightarrow A \rightarrow C \rightarrow E \rightarrow T \]

\[ \min \{5, 1, 2, 1\} = 1 \]

### Step 4

![Graph](image)

\[ S \rightarrow A \rightarrow D \rightarrow T \]

\[ \min \{4, 3, 1\} = 1 \]

**Ford-Fulkerson Algorithm**

The pseudo-path: \( S \rightarrow A \rightarrow D \leftarrow C \rightarrow T \) contains the pseudo-edge \( D \leftarrow C \) with a flow of 3. So instead of moving 3 units through the path: \( S \rightarrow B \rightarrow C \rightarrow D \rightarrow T \), we move only 1 unit and the other 2 units will reach \( T \), using the path \( S \rightarrow B \rightarrow C \rightarrow T \).

To obtain an optimal solution, we move 3 units through the path \( S \rightarrow A \rightarrow D \rightarrow T \).

### Revised Step 1

![Graph](image)

1: \( S \rightarrow B \rightarrow C \rightarrow D \rightarrow T \)

2: \( S \rightarrow B \rightarrow C \rightarrow T \)

### Revised Step 2

![Graph](image)

2: \( S \rightarrow B \rightarrow E \rightarrow T \)

### Revised Step 3

![Graph](image)

1: \( S \rightarrow A \rightarrow C \rightarrow E \rightarrow T \)

### Revised Step 4

![Graph](image)

3: \( S \rightarrow A \rightarrow D \rightarrow T \)
Using the pseudo path $S \to A \to B \to C \to T$ in the following transportation problem, we obtain the optimal Solution.

**Exercise:** Solve the following transportation problem to obtain a maximal flow:
The Assignment Problem (A Bipartite Problem)

The assignment problem deals with assigning machines to tasks, workers to jobs, applicants to positions, and so on. The goal is to determine the optimum assignment that, for example, minimizes the total cost or maximizes the team effectiveness. The assignment problem is a fundamental problem in the area of combinatorial optimization.

Assume for example that we have four jobs that need to be executed by four workers. Because each worker has different skills, the time required to perform a job depends on the worker who is assigned to it. The matrix below shows the time required (in minutes) for each combination of a worker and a job. The jobs are denoted by J1, J2, J3, and J4, the workers by W1, W2, W3, and W4.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worker 1</td>
<td>90</td>
<td>75</td>
<td>75</td>
<td>80</td>
</tr>
<tr>
<td>Worker 2</td>
<td>35</td>
<td>85</td>
<td>55</td>
<td>65</td>
</tr>
<tr>
<td>Worker 3</td>
<td>125</td>
<td>95</td>
<td>90</td>
<td>115</td>
</tr>
<tr>
<td>Worker 4</td>
<td>45</td>
<td>110</td>
<td>95</td>
<td>115</td>
</tr>
</tbody>
</table>

Each worker should perform exactly one job and the objective is to minimize the total time required to perform all jobs.

Suppose now the following table represents the qualification (scores) of 4 applicants for 4 different jobs. This time, the objective is to maximize the total scores.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>90</td>
<td>75</td>
<td>75</td>
<td>80</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>35</td>
<td>85</td>
<td>55</td>
<td>65</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>125</td>
<td>95</td>
<td>90</td>
<td>115</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>45</td>
<td>110</td>
<td>95</td>
<td>115</td>
</tr>
</tbody>
</table>

The maximum score is 125. So we subtract each score from 125 and obtain:

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>35</td>
<td>50</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>90</td>
<td>40</td>
<td>70</td>
<td>60</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>0</td>
<td>30</td>
<td>35</td>
<td>10</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>80</td>
<td>15</td>
<td>30</td>
<td>10</td>
</tr>
</tbody>
</table>

Now the problem becomes a minimization problem.

Formal mathematical definition

The formal definition of the assignment problem (or linear assignment problem) is:

Given two sets, $A$ and $T$, of equal size, together with a weight (or cost) function $C : A \times T \rightarrow \mathbb{R}$.

Find a bijection $f : A \rightarrow T$ such that the cost function:

$$\sum_{a \in A} C(a, f(a))$$

is minimized.
Hungarian Algorithm for Assignment Problem (1955)

The Hungarian algorithm can be used to find this optimal assignment. The steps of the Hungarian algorithm will be explained by the following example.

Example: Consider the 4 x 4 cost matrix:

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>90</td>
<td>75</td>
<td>75</td>
<td>80</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>35</td>
<td>85</td>
<td>55</td>
<td>65</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>125</td>
<td>95</td>
<td>90</td>
<td>115</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>45</td>
<td>110</td>
<td>95</td>
<td>115</td>
</tr>
</tbody>
</table>

Step 1. From each row, we find the row minimum. If there are more than one row minimum, we only choose one.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>0</td>
<td>50</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>35</td>
<td>5</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>0</td>
<td>65</td>
<td>50</td>
<td>70</td>
</tr>
</tbody>
</table>

Step 2. We subtract the row minimum from all entries on that row.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>0</td>
<td>50</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>35</td>
<td>5</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>0</td>
<td>65</td>
<td>50</td>
<td>70</td>
</tr>
</tbody>
</table>

Step 3. From each column, we find the column minimum. If there are more than one column minimum, we only choose one. Then we subtract the column minimum from all entries on that column

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>0</td>
<td>50</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>35</td>
<td>5</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>0</td>
<td>65</td>
<td>50</td>
<td>65</td>
</tr>
</tbody>
</table>

Step 4. We draw lines across rows and columns in such a way that all zeros are covered and that the minimum number of lines have been used (in this case lines across the 1st and the 3rd row and across the 1st column). If the number of lines just drawn is exactly n (number of rows of the cost matrix).
matrix), we are done. If not, then we go to step 5. Now the number of lines is $3 < n = 4$. If the number of lines is equal to 4, then a solution is found, otherwise we go to step 5.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>---</td>
<td>15---</td>
<td>---</td>
<td>0---</td>
</tr>
<tr>
<td>Applicant 2</td>
<td></td>
<td>0</td>
<td>50</td>
<td>20</td>
</tr>
<tr>
<td>Applicant 3</td>
<td></td>
<td>35</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 4</td>
<td></td>
<td>0</td>
<td>65</td>
<td>50</td>
</tr>
</tbody>
</table>

**Step 5.** We find the smallest entry which is not covered by the lines, which in this case is 5, and subtract it from each entry not covered by the lines. We also add it to each entry which is covered by a vertical and a horizontal line.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>---</td>
<td>20---</td>
<td>---</td>
<td>0---</td>
</tr>
<tr>
<td>Applicant 2</td>
<td></td>
<td>0</td>
<td>45</td>
<td>20</td>
</tr>
<tr>
<td>Applicant 3</td>
<td></td>
<td>35</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 4</td>
<td></td>
<td>0</td>
<td>60</td>
<td>50</td>
</tr>
</tbody>
</table>

**Step 6.** We remove all the lines and go to Step 1.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>20</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>0</td>
<td>45</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>0</td>
<td>60</td>
<td>50</td>
<td>60</td>
</tr>
</tbody>
</table>

**Step 7.** All the minimum rows and minimum columns are zero, so we must cover all the zeros by a minimum number of horizontal and vertical rows.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>---</td>
<td>20---</td>
<td>---</td>
<td>0---</td>
</tr>
<tr>
<td>Applicant 2</td>
<td></td>
<td>0</td>
<td>45</td>
<td>20</td>
</tr>
<tr>
<td>Applicant 3</td>
<td></td>
<td>35</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 4</td>
<td></td>
<td>0</td>
<td>60</td>
<td>50</td>
</tr>
</tbody>
</table>

**Step 8.** The minimum of uncovered entries is 20, so we find the next matrix:

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>---</td>
<td>40---</td>
<td>---</td>
<td>0---</td>
</tr>
<tr>
<td>Applicant 2</td>
<td></td>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 3</td>
<td></td>
<td>55---</td>
<td>---</td>
<td>0---</td>
</tr>
<tr>
<td>Applicant 4</td>
<td></td>
<td>0</td>
<td>40</td>
<td>30</td>
</tr>
</tbody>
</table>
Step 9. We remove all the lines:

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>40</td>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>0</td>
<td>40</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

Step 10. All the minimum rows and minimum columns are zero, so we must cover all the zeros by a minimum number of horizontal and vertical rows. This time the minimum number of lines is \( n = 4 \).

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>40</td>
<td>0</td>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>0</td>
<td>25</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>55</td>
<td>0</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>0</td>
<td>40</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

Step 10. Now we choose one single zero from each row and each column. Clearly the optimum solution is not always unique.

<table>
<thead>
<tr>
<th></th>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 1</td>
<td>40</td>
<td>0*</td>
<td>25</td>
<td>0*</td>
</tr>
<tr>
<td>Applicant 2</td>
<td>0</td>
<td>25</td>
<td>0*</td>
<td>0*</td>
</tr>
<tr>
<td>Applicant 3</td>
<td>55</td>
<td>0*</td>
<td>0*</td>
<td>15</td>
</tr>
<tr>
<td>Applicant 4</td>
<td>0*</td>
<td>40</td>
<td>30</td>
<td>40</td>
</tr>
</tbody>
</table>

The following table represents the job assignment:

<table>
<thead>
<tr>
<th>Applicant 1 ↔ Job 4 at a cost of 80</th>
<th>Applicant 1 ↔ Job 2 at a cost of 75</th>
</tr>
</thead>
<tbody>
<tr>
<td>Applicant 2 ↔ Job 3 at a cost of 55</td>
<td>Applicant 2 ↔ Job 4 at a cost of 65</td>
</tr>
<tr>
<td>Applicant 3 ↔ Job 2 at a cost of 95</td>
<td>Applicant 3 ↔ Job 3 at a cost of 90</td>
</tr>
<tr>
<td>Applicant 4 ↔ Job 1 at a cost of 45</td>
<td>Applicant 4 ↔ Job 1 at a cost of 45</td>
</tr>
<tr>
<td>Total minimum cost: 275</td>
<td>Total minimum cost: 275</td>
</tr>
</tbody>
</table>
Binary Relation

To define relations on sets we must have a concept of an ordered pair. To have a rigorous definition of ordered pair, we aim to satisfy one important property, namely, for sets \{a, b, c, d\} such that \((a, b) = (c, d)\) if and only if \(a = c\) and \(b = d\).

A binary relation \(R\) between the sets \(X\) (the set of departure) and \(Y\) (the set of destination) is specified by its graph \(G\), which is a subset of \(X \times Y\). Thus

\[
R = \{(x, y) : x \in X, y \in Y\} \subseteq X \times Y.
\]

The inverse of \(R\) is the set

\[
R^{-1} = \{(y, x) : y \in Y, x \in X\} \subseteq Y \times X.
\]

If \(Y = X\), then we say that \(R\) is a relation on \(X\). We have

\[
R = \{(x, y) : x \in X, y \in X\} \subseteq X \times X.
\]

An example of a binary relation is the “Division” relation between the set of prime numbers \(P\) and the set of integers \(\mathbb{Z}\), in which every prime \(p\) is associated with every integer \(z\) that is a multiple of \(p\) (but with no integer that is not a multiple of \(p\)). In this relation, for instance, the prime 2 is associated with numbers that include \(-4, 0, 4, 6, 10\), but not \(1\) or \(9\); and the prime 3 is associated with numbers that include \(0, 6,\) and \(9\), but not \(4\) or \(13\). We write \(2 R 10\) and \(3 R 9\). Notice that we have \(2 R 6\), but also \(3 R 6\).

The order of the elements in each pair of \(G\) is important: if \(a \neq b\), then \(a R b\) and \(b R a\) can be true or false, independently of each other. Resuming the above example, the prime 3 divides the integer 9, but 9 doesn’t divide 3.

The domain of \(R\) is the set of all \(x\) such that \(x R y\), for at least one \(y\). The range of \(R\) is the set of all \(y\) such that \(x R y\) for at least one \(x\).

Any binary relation on finite set may produce a matrix and also a simple or digraph.

If the sets \(X\) and \(Y\) are finite and not equal, then the corresponding graph \(G\) to \(R\) is a bipartite graph. One may also obtain a matrix from the relation.

Let \(R\) be a binary relation on a finite set \(V = \{v_1, v_2, ..., v_n\}\). We may describe the relation \(R\) by drawing a directed graph as follows:

For each element \(v_i \in V\), we draw a vertex and name it by \(v_i\). For two vertices \(v_i\) and \(v_j\), if \(v_i R v_j\) we draw an arrow from \(v_i\) to \(v_j\). When \(v_i = v_j\), the directed edge becomes a directed loop. The resulted graph is a digraph of \(R\).

Properties

Some important properties that a binary relation \(R\) over a set \(X\) may have:

**Reflexive:** For all \(x\) in \(X\) it holds that \(x R x\). For example, "greater than or equal to" (\(\geq\)) is a reflexive relation but "greater than" (\(>\)) is not.

**Irreflexive:** For all \(x\) in \(X\) it holds that not \(x R x\). For example, (\(>\)) is an irreflexive relation, but (\(\geq\)) is not.
Symmetric: For all $x$ and $y$ in $X$ it holds that if $x \mathcal{R} y$, then $y \mathcal{R} x$. The relation "is equal to" is symmetric, but "is less than" is not symmetric. Note that a symmetric relation, $\mathcal{R}^T = \mathcal{R}$.

Anti-symmetric: For all $x$ and $y$ in $X$, if $x \mathcal{R} y$ and $y \mathcal{R} x$, then $x = y$. For example, the relation ($\geq$) is anti-symmetric, so is ($>$).

Asymmetric: For all $x$ and $y$ in $X$, if $x \mathcal{R} y$ then not $y \mathcal{R} x$. A relation is asymmetric if and only if it is both anti-symmetric and irreflexive. For example, ($>$) is asymmetric, but ($\geq$) is not.

Transitive: for all $x, y$ and $z$ in $X$ it holds that if $x \mathcal{R} y$ and $y \mathcal{R} z$ then $x \mathcal{R} z$. For example, ($>$) and ($\geq$) are transitive.

Connectivity relation: Let $\mathcal{R}$ be a relation on a set $\mathcal{P}$. The connectivity relation $\mathcal{R}^*$ consists of the pairs $(a, b)$ such that there is a path of length at least one from $a$ to $b$ in $\mathcal{R}$.

$\mathcal{R}^n$ consists of the pairs $(a, b)$ such that there is a path of length $n$ from $a$ to $b$, it follows that $\mathcal{R}^*$ is the union of all the sets $\mathcal{R}^n$. In other words,

$$\mathcal{R}^* = \bigcup_{n=1}^{\infty} \mathcal{R}^n$$

Example: Let $\mathcal{R}$ be the relation on the set of all states in the United States that contains $(a, b)$ if state $a$ and state $b$ have a common border. The relation $\mathcal{R}^n$ consists of the pairs $(a, b)$, where it is possible to go from state $a$ to state $b$, by crossing exactly $n$ state borders. $\mathcal{R}^*$ consists of the ordered pairs $(a, b)$, where it is possible to go from state $a$ to state $b$ crossing as many borders as necessary. The only ordered pairs not in $\mathcal{R}^*$ are those containing states that are not connected to the continental United States (i.e., those pairs containing Alaska or Hawaii).

The transitive closure of a relation $\mathcal{R}$ equals the connectivity relation $\mathcal{R}^*$.

Comparability (trichotomy law): For any $x$ and $y$ in $X$, either $x \mathcal{R} y$ or $y \mathcal{R} x$. The relations ($>$) and ($\geq$) on the real line $\mathbb{R}$ satisfy this condition.

Definitions:

A given binary relation $\mathcal{R}$ on a set $X$ is said to be an equivalence relation, if it is reflexive, symmetric and transitive. For example, the relation defined as: “$x \mathcal{R} y$, if and only if $x$ and $y$ are both divisible by 5,” is an equivalence relation.

A partial order is a binary relation $\mathcal{R}$ over a set $\mathcal{P}$ which is reflexive, antisymmetric, and transitive. A set with a partial order is called a partially ordered set (also called a poset). In a poset elements can’t be comparable.
The set $\mathbb{R} \times \mathbb{R}$ is a partially ordered set for $(\forall)$, since $(2,3)$ and $(3,2)$ cannot be compared. The divisibility relation | is partial ordering on the set of positive integers $\mathbb{Z}^+$, because it is reflexive, antisymmetric, and transitive, so $(\mathbb{Z}^+, |)$ is a poset. But $(\mathbb{Z}^+, |)$.

Note: Let $x \, R \, y$ be the relation on the set of people such that $x \, R \, y$, if $x$ and $y$ are people and $x$ is older than $y$. $R$ is antisymmetric because if a person $x$ is older than a person $y$, then $y$ is not older than $x$. That is, if $x$ is older than $y$, then $y$ is not older than $x$. Thus $y \, R \, x$ is not true. The relation $R$ is transitive because if person $x$ is older than person $y$ and $y$ is older than person $z$, then $x$ is older than $z$. That is, if $x \, R \, y$ and $y \, R \, z$, then $x \, R \, z$. However, $R$ is not reflexive, because no person is older than himself or herself. It follows that $R$ is not a partial ordering. But the relation is comparable.

A totally ordered set or "linearly ordered set" is a set plus a relation on the set (called a total order) that satisfies the conditions for a partial order plus the comparability condition. The set of real line $\mathbb{R}$ is a totally ordered set for $(\forall)$. Also any two real numbers are comparable.

Greatest element and least element of poset $(P, R)$:

(i) An element $G$ in $P$ is a greatest element if for every element $x$ in $P$, $a \, R \, G$.

(ii) An element $L$ in $P$ is a least element if for every element $x$ in $P$, $L \, R \, x$.

A poset can only have one greatest or least element.

$(P, R)$ is a well-ordered set, if it is a poset such that $R$ is a total ordering and every nonempty subset of $P$ has a least element.

The set of ordered pairs of positive integers, $\mathbb{Z}^+ \times \mathbb{Z}^+$, with

$$(a, b) < (c, d), \text{ if } a < c \text{ or if } a = c \text{ and } b < d$$

(the lexicographic ordering), is a well-ordered set.

Example: The fact that $3 < 4$, we have $(3, 5) < (4, 8)$ and also $(3, 8) < (4, 5)$. We have $(4, 9) < (4, 11)$, because the first entries of $(4, 9)$ and $(4, 11)$ are the same and $9 < 11$. Note that $(1, 2, 3, 5) < (1, 2, 4, 3)$, because the entries in the first two positions of these 4-tuples agree, but in the third position the entry in the first 4-tuple, 3, is less than that in the second 4-tuple, 4. (Here the ordering on 4-tuples is the lexicographic ordering that comes from the usual “less than or equals” relation on the set of integers.)

The set $\mathbb{Z}$, with the usual $\leq$ ordering, is not well-ordered because the set of negative integers, which is a subset of $\mathbb{Z}$, has no least element.

The words in a dictionary are listed in alphabetic, or lexicographic, order, which is based on the ordering of the letters in the alphabet.
Maximal and minimal of poset $(P, R)$:

(i) An element $M$ in $P$ is maximal, if there is no element $x$ in $P$ such that $M R x$.

(ii) An element $m$ in $P$ is maximal, if there is no element $x$ in $P$ such that $x R M$.

Maximal and minimal elements are easy to spot using a Hasse diagram (which will be discussed later). They are the “top” and “bottom” elements in the diagram.

Upper and lower bounds of poset $(P, R)$: For a subset $A$ of $P$, an element $x$ in $P$ is an upper bound of $A$ if $a R x$, for each element $a$ in $A$. In particular, $x$ need not be in $A$ to be an upper bound of $A$. Similarly, an element $y$ in $P$ is a lower bound of $A$ if $y R a$, for each element $a$ in $A$.

A greatest element of $P$ is an upper bound of $P$ itself, and a least element is a lower bound of $P$.

Closure

Consider the relation $R = \{(a, d), (b, a), (b, c), (c, a), (c, d), (d, c)\}$ with the matrix

$$M_R = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

To obtain reflexive closure we need to add the set $\{(a, a), (b, b), (c, c), (d, d)\}$ to $R$. That is adding 1 to the diagonal.

To obtain symmetric closure we need to make the matrix symmetric. That means we need to add the set $\{(a, b), (a, c), (c, b), (d, a)\}$ to $R$.

To obtain the transitive closure, we proceed as follows:

1. Since we have $(a, c)$ and $(c, a)$ we must add $(a, a)$ to the relation $R$.
2. We also have $(a, d)$ and $(d, c)$, so we need to include $(a, c)$ to the relation $R$.
3. From $(b, a)$ and $(a, c)$, we include $(b, c)$ to the relation $R$.
4. From $(b, c)$ and $(c, a)$, we include $(b, a)$ to the relation $R$.
5. From $(b, c)$ and $(c, d)$, we include $(b, d)$ to the relation $R$.
6. From $(c, a)$ and $(c, a)$, we include $(c, c)$ to the relation $R$.
7. From $(d, c)$ and $(c, d)$, we include $(d, d)$ to the relation $R$.
8. From $(d, c)$ and $(c, a)$, we include $(d, c)$ to the relation $R$.

Hence $R^* = R \cup R_0$, where

$$R_0 = \{(a, a), (a, c), (b, c), (b, a), (b, d), (c, c), (d, d), (d, c)\}.$$
Lattice Theory

A lattice consists of a partially ordered set \((L, R)\) in which each two-element subset \(\{a, b\}\) has a unique supremum (also called a least upper bound or join), denoted \(\operatorname{sup} \{a, b\}\) and a unique infimum (also called a greatest lower bound or meet), denoted \(\operatorname{inf} \{a, b\}\).

A bounded lattice \((L, R)\) is a lattice that additionally has a greatest element \(1\) and a least element \(0\), which satisfy:

\[ 0 \ R x \ R 1, \text{ for every } x \text{ in } L. \]

The greatest and least element are also called the maximum and minimum, or the top and bottom elements.

A lattice element \(y\) is said to cover another element \(x \neq y\), if, but there does not exist a \(z\) such that \(x \ R z \ R y\).

An example is given by the natural numbers, partially ordered by divisibility, for which the unique supremum is the least common multiple and the unique infimum is the greatest common divisor.

Lattice of integer divisors of 60

If two posets have the same lattice, then we say that they are isomorphic. Here is an example:

Lattice of integer divisors of 30

Lattice of subsets of \(\{x, y, z\}\)

Lattices have many special properties. Furthermore, lattices are used in many different applications such as models of information flow and play an important role in Boolean algebra.
Hasse Diagrams

Like relations and functions, partial orders have a convenient graphical representation, known as Hasse Diagrams. Consider the digraph representation of a partial order. Because we are dealing with a partial order, we know that the relation must be reflexive and transitive. Thus, we can simplify the graph as follows:

- Remove all self-loops;
- Remove all transitive edges;
- Remove directions on edges assuming that they are oriented upwards.

The resulting diagram is far simpler.

**Formal definition:** A graphical rendering of a partially ordered set (poset) \((P, R)\) displayed via the cover relation of the poset with an implied upward orientation. A point is drawn for each element of the poset, and line segments are drawn between these points according to the following two rules:

1. If \(x \mathrel{R} y\) in the poset, then the point corresponding to \(x\) appears lower in the drawing than the point corresponding to \(y\).
2. The line segment between the points corresponding to any two elements \(x\) and \(y\) of the poset is included in the drawing if and only if \(x\) covers \(y\) or \(y\) covers \(x\).

Consider the partially ordered set \((P, R)\), where \(P = \{1, 2, 3, 4, 5, 6, 7, 8\}\) and \(R\) is the divisibility relation. Here is how we change the digraph into the Hasse diagram

**Examples:** Draw a Hasse diagram for \((P)\) (divisibility relation).
\[ P_1 = \{1, 2, 3, 6, 12, 24\} \]

\[ P_2 = \{2, 3, 4, 5, 6, 30, 60\} \]

\[ P_3 = \{1, 2, 3, 5, 6, 10, 15, 30\} \]

\[ P_4 = \{2, 4, 6, 12, 24, 36\} \]

The same as the lattice

Constructing the Hasse Diagram of \( P = \{1, 2, 3, 4, 6, 8, 12\} \) and \( R \) is the divisibility relation:

Notice that 8 and 12 on the top are the maximal elements and 1 on the bottom is the minimal element of the poset \( P \). Also 1 is the least element; but \( P \) has no greatest element.