Elementary Graph Algorithms

**What is a Graph?**

- Graph is a diagram representing a system of connections or interrelations among two or more things (objects). The interconnected objects are represented by mathematical abstractions called vertices (or nodes), and the links that connect some pairs of vertices are called edges. Typically, a graph is depicted in diagrammatic form as a set of dots (circles) for the vertices, joined by lines or curves for the edges.

- There are certain intuitive things one can easily recognize. Remember that a relationship can be symmetric or antisymmetric (e.g. “friend of” is a symmetric one and “mother of” is antisymmetric). The nature of the relationship we are modeling says if the graph is undirected or directed.

- Example: Consider two relationships (where the objects are persons): “shook hands with” and “knows of”; what kind of relationship is each??

- Look at some more pictures of more complicated graphs in real life (most or all of you have already seen some of these in your everyday life).
Directed & Undirected Graphs

A graph is usually specified as a pair \( G = (V, E) \) where \( V = \{v_1, v_2, \ldots, v_n\} \) is called the set of nodes (\( n \) is the number of nodes) and \( E = \{e_1, e_2, \ldots, e_m\} \) is the set of edges (\( m \) is the number of edges).

Sometimes, an edge \( e \) is also denoted by a pair of vertices \((v_i, v_j)\) where \( v_i \) and \( v_j \) are the two end points of the edge. If the pair is an ordered pair, the edge is a directed edge, from \( v_i \) to \( v_j \); otherwise, the edge is undirected. If all edges are directed, the graph is called a directed graph; if all edges are undirected, the graph is called an undirected graph.

\[
V = \{1, 2, 3, 4, 5, 6, 7, 8\} \\
E = \{1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, \ \\
5-6, 7-8\} \\
m = 11, n = 8
\]

Computational Representation

There are two simple standard ways to represent a graph; either way applies to both undirected and directed graphs:

- A collection of adjacency lists: a linked list of edges for each vertex – efficient for sparse graphs – each edge is represented twice for undirected graphs and once for directed graphs.
- An adjacency matrix – an \( |V| \times |V| \) Boolean matrix [2-D array] where 1 denotes existence of an edge and 0 denotes its absence. \( A_{ij} = 1 \) says that the edge \((i, j)\) exists.

- Nodes are usually numbered 1 through \( n \) in any arbitrary manner; edges are usually referred to as node pairs.
- In adjacency listing ordering of nodes in the lists does not matter in simple applications.

Usually, \( |V| = n, |E| = m \). For our purpose, we will not consider self-loops.

Adjacency matrix is symmetric for undirected (symmetric) graphs, i.e., \( A_{ij} = A_{ji} \) for all \( i,j \).

Quiz: What is the maximum possible value of \( m \) for a given \( n \)? \( \frac{n(n-1)}{2} \) Why?
### Some graph applications

<table>
<thead>
<tr>
<th>graph</th>
<th>node</th>
<th>edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>communication</td>
<td>telephone, computer</td>
<td>fiber optic cable</td>
</tr>
<tr>
<td>circuit</td>
<td>gate, register, processor</td>
<td>wire</td>
</tr>
<tr>
<td>mechanical</td>
<td>joint</td>
<td>rod, beam, spring</td>
</tr>
<tr>
<td>financial</td>
<td>stock, currency</td>
<td>transactions</td>
</tr>
<tr>
<td>transportation</td>
<td>street intersection, airport</td>
<td>highway, airway route</td>
</tr>
<tr>
<td>internet</td>
<td>class C network</td>
<td>connection</td>
</tr>
<tr>
<td>game</td>
<td>board position</td>
<td>legal move</td>
</tr>
<tr>
<td>social relationship</td>
<td>person, actor</td>
<td>friendship, movie cast</td>
</tr>
<tr>
<td>neural network</td>
<td>neuron</td>
<td>synapse</td>
</tr>
<tr>
<td>protein network</td>
<td>protein</td>
<td>protein-protein interaction</td>
</tr>
<tr>
<td>molecule</td>
<td>atom</td>
<td>bond</td>
</tr>
</tbody>
</table>

### Graph Representation: Adjacency Matrix

Graph representation: adjacency matrix

**Adjacency matrix.** An $n$-by-$n$ matrix with $A_{uv} = 1$ if $(u, v)$ is an edge.

- Two representations of each edge.
- Space proportional to $n^2$.
- Checking if $(u, v)$ is an edge takes $O(1)$ time.
- Identifying all edges takes $O(n^2)$ time.

![Adjacency Matrix Image]

```
1 0 1 1 0 0 0 0
2 0 1 0 1 1 0 0 0
3 1 0 0 1 0 1 1
4 0 1 0 0 1 0 0 0
5 0 1 1 1 0 1 0 0
6 0 0 0 0 1 0 0 0
7 0 0 1 0 0 0 1 0
8 0 0 0 0 1 0 0 1
```
**Graph Representation: Adjacency Lists**

**Adjacency lists.** Node-indexed array of lists.
- Two representations of each edge.
- Space is $\Theta(m + n)$.
- Checking if $(u, v)$ is an edge takes $O(\text{degree}(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.

**Paths and connectivity**

**Definition.** A path in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_k$ with the property that each consecutive pair $v_{i-1}, v_i$ is joined by a different edge in $E$.

**Definition.** A path is **simple** if all nodes are distinct.

**Definition.** An undirected graph is **connected** if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$. 
**Cycles and Trees**

- **Definition.** A **cycle** is a path $v_1, v_2, \ldots, v_k$ in which $v_1 = v_k$ and $k \geq 2$.
- **Definition.** A cycle is **simple** if all nodes are distinct (except for $v_1$ and $v_k$).
- **Definition.** An undirected graph is a **tree** if it is connected and does not contain a cycle.

**Theorem.** Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third: (1) $G$ is connected; (2) $G$ does not contain a cycle; (3) $G$ has $n - 1$ edges.

![Cycle C = 1-2-4-5-3-1](image)

---

**Rooted Trees & Connectivity**

- Given a tree $T$, choose a **root** node $r$ and orient each edge away from $r$.
- **Importance.** Models hierarchical structure.

![Rooted Trees](image)

- **$s$-$t$ connectivity problem.** Given two nodes $s$ and $t$, is there a path between $s$ and $t$?
- **$s$-$t$ shortest path problem.** Given two nodes $s$ and $t$, what is the length of a shortest path between $s$ and $t$?
- **Applications.** Friendster, Maze traversal, Kevin Bacon number, Fewest hops in a communication network.
Some simple concepts

Applications: any network (computer network, social network, financial networks, …), modeling, and many many others.

Which representation to choose? Depends on the nature of application. Adjacency matrix takes \(O(n^2)\) storage and adjacency listing takes \(O(nm)\) storage.

In degrees and out degrees: the number of edges incident on a node (distinction is made between incoming and out going edges in directed graphs).

How to represent vertex and edge attributes? Easy – use weight vectors.

Choice of 0 or \(\infty\) for the entries in adjacency matrix to denote absence: depends on application: we will use 0 in simple applications.

Connected components: Path between 2 nodes, Reachability, Connected (undirected graphs) and strongly connected (directed graph).

Incidence matrix

The incidence matrix of a directed graph \(G = (V, E)\) with no self-loops is a \(|V| \times |E|\) matrix \(B = (b_{ij})\) such that

\[
\begin{cases}
-1 & \text{if edge } j \text{ leaves vertex } i \\
1 & \text{if edge } j \text{ enters vertex } i \\
0 & \text{otherwise}
\end{cases}
\]

Representing Attributes, e.g., edge weights, node weights, and others.

Spanning Tree of a connected graph \(G\) is a subgraph of \(G\) such that it is connected but acyclic.

Question: Describe what the entries of the matrix product \(BB^T\) represent, where \(B^T\) is the transpose of \(B\).

More simple concepts

Incidence Matrix of a directed graph: Let \(G\) be a graph with \(V(G) = \{1, \ldots, n\}\) and \(E(G) = \{e_1, \ldots, e_m\}\), where each edge has a direction.

The incidence matrix of a directed graph \(G = (V, E)\) with no self-loops is a \(|V| \times |E|\) matrix \(B = (b_{ij})\) such that

\[
\begin{cases}
-1 & \text{if edge } j \text{ leaves vertex } i \\
1 & \text{if edge } j \text{ enters vertex } i \\
0 & \text{otherwise}
\end{cases}
\]

Example:

\[
\begin{bmatrix}
-1 & 1 & 0 & 0 & 0 \\
1 & 0 & -1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & -1
\end{bmatrix}
\]

\[
\begin{bmatrix}
-1 & 1 & 0 & 0 & 0 \\
1 & 0 & -1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 1 & -1
\end{bmatrix}
\]

Design algorithms to compute \(B\) from \(A\) and also the product \(BB^T\). Describe what the entries of the matrix product \(BB^T\) represent, where \(B^T\) is the transpose of \(B\).

Incidence matrix of an undirected graph is defined similarly (– 1’s are replaced by 1s)

The square of a directed graph \(G = (V, E)\) is the graph \(G^2 = (V, E^2)\) such that (u, v) \(\in E^2\) if and only \(\ G\) contains a path with at most two edges between \(u\) and \(v\). Describe efficient algorithms for computing \(G^2\) from \(G\) for both the adjacency list and adjacency-matrix representations of \(G\). Analyze the running times of your algorithms.
Some Very Easy Exercise problems

- The **square** of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only if $G$ contains a path with at most two edges between $u$ and $v$. Describe efficient algorithms for computing $G^2$ from $G$ for both the adjacency list and adjacency-matrix representations of $G$. Analyze the running times of your algorithms.

- The **transpose** of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, $G^T$ is $G$ with all its edges reversed. Describe efficient algorithms for computing $G^T$ from $G$, for both the adjacency list and adjacency-matrix representations of $G$. Analyze the running times of your algorithms.

Graph Traversal

- The most basic graph algorithm that visits nodes of a graph in certain order
- Used as a subroutine in many other algorithms
- Two Simple Traversals: (Assume symmetric or undirected graphs)
  - Depth-First Search (DFS): uses recursion (stack)
  - Breadth-First Search (BFS): uses queue
- **DFS(v):** visits all the nodes reachable from the given node $v$ in depth-first order
  - Mark $v$ as visited (how?)
  - For each edge $v \rightarrow u$: If $u$ is not visited, call DFS($u$).
  - **Note:** non-recursive version replaces recursive calls with a stack (user maintained)
- **BFS(v):** visits all the nodes reachable from $v$ in breadth-first order
  - Initialize a queue $Q$
  - Mark $v$ as visited and push it to $Q$
  - While $Q$ is not empty:
    - Take the front element of $Q$ and call it $w$
    - For each edge $w \rightarrow u$: If $u$ is not visited, mark it as visited and push it to $Q
**Breadth First Search (BFS)**

- **Breadth-first search** is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms.
- **BFS intuition.** Explore outward from s in all possible directions, adding nodes one “layer” at a time.

![Diagram](image)

- **Algorithm:**
  1. \( L_0 = \{ s \} \).
  2. \( L_1 = \) all neighbors of \( L_0 \).
  3. \( L_2 = \) all nodes that do not belong to \( L_0 \) or \( L_1 \), and that have an edge to a node in \( L_1 \).
  4. \( L_{i+1} = \) all nodes that do not belong to an earlier layer, and that have an edge to a node in \( L_i \).

- **Theorem:** For each \( i \), \( L_i \) consists of all nodes at distance exactly \( i \) from \( s \). There is a path from \( s \) to \( t \) iff \( t \) appears in some layer.

---

**BFS Details**

- Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- The algorithm discovers all vertices at distance \( k \) from \( s \) before discovering any vertices at distance \( k + 1 \).
- **BFS colors each vertex white, gray, or black.** All vertices start out white A vertex is discovered the first time it is encountered during the search.
- BFS distinguishes between them to ensure that the search proceeds in a breadth-first manner.
- If \((u,v) \in E\) and vertex \( u \) is black, then vertex \( v \) is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.
- We assume the graph is input by its adjacency lists.
- We store the color of each vertex \( u \in V \) in the attribute \( u.\text{color} \) and the predecessor of \( u \) in the attribute \( u.\pi \). If \( u \) has no predecessor (for example, if \( u = s \) or \( u \) has not been discovered), then \( u.\pi = \text{NIL} \).
- The attribute \( u.d \) holds the distance from the source \( s \) to vertex \( u \) computed by the algorithm. The algorithm also uses a first-in, first-out queue \( Q \) to manage the set of gray vertices.
**BFS Algorithm & Execution Example**

The operation of BFS on an undirected graph. Tree edges are shown shaded as they are produced by BFS. The value of $u.d$ appears within each vertex $u$. The queue $Q$ is shown at the beginning of each iteration of the while loop of lines 10–18. Vertex distances appear below vertices in the queue.

```
BFS(G, s)
1  for each vertex $u \in G.V - \{s\}$
2      $u.color = \text{WHITE}$
3      $u.d = \infty$
4      $u.\pi = \text{NIL}$
5      $s.color = \text{GRAY}$
6      $s.d = 0$
7      $s.\pi = \text{NIL}$
8      $Q = \emptyset$
9      \text{ENQUEUE}(Q, s)$
10     \text{while } Q \neq \emptyset$
11         \text{u = DEQUEUE}(Q)$
12         for each $v \in G.Adj[u]$
13             if $v.\text{color} = \text{WHITE}$
14                 $v.\text{color} = \text{GRAY}$
15                 $v.d = u.d + 1$
16                 $v.\pi = u$
17                 \text{ENQUEUE}(Q, v)$
18         $u.\text{color} = \text{BLACK}$
```

**Depth First Search (DFS)**

- The strategy followed by depth-first search is to search “deeper” in the graph whenever possible.
- Depth-first search explores edges out of the most recently discovered vertex $v$ that still has unexplored edges leaving it.
- Once all of $v$’s edges have been explored, the search “backtracks” to explore edges leaving the vertex from which $v$ was discovered.
- This process continues until we have discovered all the vertices that are reachable from the original source vertex.
- If any undiscovered vertices remain, then depth-first search selects one of them as a new source, and it repeats the search from that source.
- The algorithm repeats this entire process until it has discovered every vertex.
**DFS Details**

Whenever DFS discovers a vertex \( v \) during a scan of the adjacency list of an already discovered vertex \( u \), it records this event by setting \( v \)'s predecessor attribute \( v.\pi \) to \( u \).

Unlike BFS, a DFS may be composed of several trees, because the search may repeat from multiple sources. We define the predecessor subgraph of a depth-first search as follows: we let \( G_{\pi} = (V, E_{\pi}) \), where \( E_{\pi} = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq \text{NIL}\} \)

The predecessor subgraph of a depth-first search forms a depth-first forest comprising several depth-first trees. The edges in \( E_{\pi} \) are tree edges.

Each vertex is initially white, is grayed when it is discovered in the search, and is blackened when it is finished, that is, when its adjacency list has been examined completely. This technique guarantees that each vertex ends up in exactly one depth-first tree, so that these trees are disjoint.

The following pseudocode is the basic depth-first-search algorithm. The input graph \( G \) may be undirected or directed.

**DFS Pseudocode (Recursive)**

```
DFS(G)
1. for each vertex \( u \in G.V \)
2. \( u.\text{color} = \text{WHITE} \)
3. \( u.\pi = \text{NIL} \)
4. for each vertex \( u \in G.V \)
5. if \( u.\text{color} = \text{WHITE} \)
6. DFS-\text{VISIT}(G, u)

DFS-\text{VISIT}(G, u)
1. \( u.\text{color} = \text{GRAY} \)
2. for each \( v \in G.\text{Adj}[u] \) // explore edge \((u, v)\) //
3. if \( v.\text{color} = \text{WHITE} \)
4. \( v.\pi = u \)
5. DFS-\text{VISIT}(G, v)
6. \( u.\text{color} = \text{BLACK} \) // blacken \( u \); it is done
```

**Exercise:** Write a non-recursive version. Consider the following graph as an example.

![Graph Example](image-url)
A Step by Step Example Execution of DFS

Assume that each edge of a graph is associated with a weight function \( w: E \rightarrow \mathbb{R}^+ \) (only non-negative weights). For simplicity, assume that all edge weights are distinct (this is not necessary).

Shortest Path Problem: Given a source node, say \( s \), and a destination node, say \( t \), find the shortest (cost of the path is the sum of the costs of all edges lying on the path) path from \( s \) to \( t \).

Example:

**Approach:** We start from source \( s \); we know the nodes directly reachable from \( s \); none of them is the destination \( t \) in our example. Think of 2 things: (1) we have not seen \( t \) yet; the path from \( s \) to \( t \) must pass through one of the neighbors of \( s \); (2) we sure need to explore the entire graph to compute the shortest path. [Note that the discussion applies equally well to both directed and undirected graphs with minimal changes].

---

Shortest Path(s) in a Graph

Assume that each edge of a graph is associated with a weight function \( w: E \rightarrow \mathbb{R}^+ \) (only non-negative weights). For simplicity, assume that all edge weights are distinct (this is not necessary).

Shortest Path Problem: Given a source node, say \( s \), and a destination node, say \( t \), find the shortest (cost of the path is the sum of the costs of all edges lying on the path) path from \( s \) to \( t \).

**Example:**

- Cost of the shortest path \( P \) from \( s \) to \( t \) = \( w(P) = 9 + 23 + 2 + 16 = 50 \), where \( P = (s \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow t) \).
- It is easy to see that there may exist more than one shortest paths.
- We are interested to find one such shortest path from \( s \) to \( t \).

**Approach:** We start from source \( s \); we know the nodes directly reachable from \( s \); none of them is the destination \( t \) in our example. Think of 2 things: (1) we have not seen \( t \) yet; the path from \( s \) to \( t \) must pass through one of the neighbors of \( s \); (2) we sure need to explore the entire graph to compute the shortest path. [Note that the discussion applies equally well to both directed and undirected graphs with minimal changes].
**Dijkstra's Algorithm**

- Maintain a set of **explored nodes** $S$ for which we have determined the shortest path distance $d(u)$ from $s$ to $u$.
- Initialize $S = \{ s \}$, $d(s) = 0$.
- While $S \neq V$
  - Select a node $v \not\in S$ with at least one edge from $S$ for which $\pi(v) = \min_{e=(u,v), u \in S} d(u) + w(e)$ is as small as possible and set $S = S \cup \{ v \} + w(e)$ and $d(v) = \pi(v)$.

To produce the $s-u$ paths corresponding to the distances found, we simply record the edge $(u, v)$ on which it achieved the value $\pi(v) = \min_{e=(u,v), u \in S} d(u) + w(e)$ [simple recursive backtracking will do]. Consider the previous graph; initially, the situation is as follows [the shaded area is $S$].

---

**Dijkstra's Algorithm: Example Execution**
**Dijkstra's Algorithm: Proof of Correctness**

The **Invariant** is: For each node \( u \in S \), \( d(u) \) is the length of the shortest \( s-u \) path \( [s \text{ is the given source node}] \).

We will prove by induction on \(|S|\).

- **Base case:** Initially, \( S = \{s\} \), or \(|S| = 1\); the claim is trivially true.
- **Inductive hypothesis:** Assume true for \(|S| = k \geq 1\)
  - Let \( v \) be next node added to \( S \), and let \( u-v \) be the chosen edge.
  - The shortest \( s-u \) path plus \( (u, v) \) is an \( s-v \) path of length \( \pi(v) \).
  - Consider any \( s-v \) path \( P \). We'll see that it's no shorter than \( \pi(v) \).
  - Let \( x-y \) be the first edge in \( P \) that leaves \( S \), and let \( P' \) be the subpath to \( x \); \( P \) is already too long as soon as it leaves \( S \).

\[
\begin{align*}
w(P) & \geq w(P') + w(x,y) \\
& \geq d(x) + w(x,y) \\
& \geq \pi(y) \\
& \geq \pi(v).
\end{align*}
\]

\( w \) is non-negative, \( \pi \) is defined, \( v \) is chosen over \( y \)

---

**Dijkstra's Algorithm: Implementation**

There are \( n - 1 \) iterations of the while loop for a graph with \( n \) nodes, as each iteration adds a new node \( v \) to \( S \).

- We explicitly maintain the values of the minima \( \pi(v) = \min_{e=(u,v) : u \in S} d(u) + w(e) \) for each node \( v \in V - S \); we maintain this information in a min-binary_heap [we have seen this simple data structure before]; select_min operation is done in \( O(\log n) \) worst case time, \( n \) is the number of elements in the heap.
- Next node to explore = node with minimum \( \pi(v) \).
- When exploring \( v \), for each incident edge \( e = (v, w) \), update \( \pi(w) = \min \{ \pi(w), \pi(v)+w(e) \} \)
- We get an implementation with \( O(m \log n) \) worst case time.
- We can get a better amortized time of \( O(m \log_{\log n} n) \) if we use a Fibonacci heap.
**Dijkstra's Algorithm: Implementation**

DIJKSTRA (V, E, s)
Create an empty priority queue.
FOR EACH v ≠ s : d(v) ← ∞; d(s) ← 0.
FOR EACH v ∈ V : insert v with key d(v) into priority queue.
WHILE (the priority queue is not empty)
    u ← delete-min from priority queue.
    FOR EACH edge (u, v) ∈ E leaving u:
        IF d(v) > d(u) + w(u, v)
            decrease-key of v to d(u) + w(u, v) in priority queue.
            d(v) ← d(u) + w(u, v).

---

**Topological Sort**

- **Input**: a DAG G = (V, E)
- **Output**: an ordering of nodes such that for each edge u → v, u comes before v
- **Explain Precedence Graph.**
- **Note**: There can be multiple answers. e.g., both {6, 1, 3, 2, 7, 4, 5, 8} and {1, 6, 2, 3, 4, 5, 7, 8} (and others) are valid orderings for the graph below.
Spanning Tree Properties

Consider a connected, undirected graph $G = (V, E)$; Let $T = (V, F), F \subseteq E$, be a subgraph of $G = (V, E)$. All of the following statements must be true:

1. $T$ is a spanning tree of $G$.
2. $T$ is acyclic and connected.
3. $T$ is connected and has $n - 1$ edges.
4. $T$ is acyclic and has $n - 1$ edges.
5. $T$ is minimally connected: removal of any edge disconnects it.
6. $T$ is maximally acyclic: addition of any edge creates a cycle.
7. $T$ has a unique simple path between every pair of nodes.

The black edges constitute the spanning tree. Satisfy yourself that all those statements are true [some are equivalent]

Minimal Spanning Tree

A connected, undirected graph $G = (V, E)$ with weight function $w: E \rightarrow \mathbb{R}$ (set of reals); we'll assume all edge weights are distinct [Explain briefly the general case when duplicates are allowed]

A connected acyclic subgraph $T$ of $G$ that spans all vertices is called a spanning tree of the graph. A minimal spanning tree (MST) is a spanning tree of minimum weight, i.e., a spanning tree whose sum of edge weights is minimized.

Note: Cayley’s theorem. There are $n^{n-2}$ spanning trees of $K_n$; thus, not possible to find MST by exhaustive enumeration.
Applications

- MST is fundamental problem with diverse applications.
- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

Greedy Algorithms for MST

1. **Kruskal’s algorithm.** Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

2. **Reverse-Delete algorithm.** Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$.

3. **Prim’s algorithm.** Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the least weight edge $e$ to $T$ that has exactly one endpoint in $T$.

We’ll do 1 and 3. The greedy structure is the same; implementations are different, one uses the min-heaps and the other uses Union-Find data structure. Both works in $O(m \log n)$ time.

- **Cut property.** Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e$ [cheapest way to connect $S$ and $V - S$].

- **Cycle property.** Let $C$ be any cycle, and let $f$ be the max cost edge belonging to $C$. Then the MST does not contain $f$. [Why? Assume $f$ belongs to an MST $T_1$ ⇒ deleting $f$ breaks $T_1$ in two subtrees with the two ends of $f$ in different subtrees ⇒ remainder of $C$ reconnects the subtrees, hence there is an edge $e$ of $C$ with ends in different subtrees, i.e., it reconnects the subtrees into a tree $T_2$ with weight less than that of $T_1$, because the weight of $e$ is less than the weight of $f$.]
A cut is a partition of the nodes into two nonempty subsets $S$ and $V - S$. 

Cycle: Consider an undirected graph $G$. A cycle is a simple (no node is visited more than once) path (sequence of edges) from any node back to itself.

Examples: (1) (1, 2, 3, 4, 5, 6, 1), (2) (1, 6, 7, 1), (3) (3, 4, 5, 6, 3), so on.

A cut $C = (S, T)$ is a partition of $V$ of a graph $G=(V, E)$ into two subsets $S$ and $T$. The cut-set of a cut $C=(S, T)$ is the set $\{(u,v) \in E \mid u \in S, v \in T\}$ of edges that have one endpoint in $S$ and the other endpoint in $T$.

Example: Cut $S = \{4, 5, 8\}$, $T = V - S = \{1, 2, 3, 6, 7\}$.

Cut-set $= \{5-6, 5-7, 3-4, 3-5, 7-8\}$.
Prim’s Algorithm

**Algorithm**: Initialize $S = \text{any node}$, Sort the edges in $O(m \log n)$ time, $n = \text{no. of nodes}$ and $m = \text{no. of edges}$. Then repeat $n - 1$ times: (Add to tree the min weight edge with one endpoint in $S$; Add new node to $S$.)

**Theorem**: Prim’s algorithm computes the MST. **Proof**: Special case of greedy algorithm (blue rule repeatedly applied to $S$).

**Implement**: $d(v) =$ weight of cheapest known edge between $v$ and $S$] The data structure and approach are almost identical to Dijkstra’s algorithm for shortest path.

PRIM $(V, E, w)$
Create an empty priority queue.
$s \leftarrow \text{any node in } V.$
FOR EACH $v : d(v) \leftarrow \infty; d(s) \leftarrow 0.$
FOR EACH $v$ : insert $v$ with key $d(v)$ into priority queue.
REPEAT $n - 1$ times
  $u \leftarrow \text{delete-min from priority queue}.$
  FOR EACH edge $(u, v) \in E$ incident to $u$:
    IF $d(v) > w(u, v)$
      decrease-key of $v$ to $w(u, v)$ in priority queue.
    $d(v) \leftarrow w(u, v).$
**Kruskal's algorithm**

Consider edges in ascending order of weight:

**Algorithm:** Add to tree unless it would create a cycle.

**Theorem.** Kruskal's algorithm computes the MST.

**Proof.** Special case of greedy algorithm.

- Case 1: both endpoints of \( e \) in same blue tree.
  \[ \Rightarrow \text{color red by applying red rule to unique cycle} \]
- Case 2. If both endpoints of \( e \) are in different blue trees.
  \[ \Rightarrow \text{color blue by applying blue rule to cut-set defined by either tree} \]

**Kruskal’s Algorithm: implementation**

**Theorem.** Kruskal's algorithm can be implemented in \( O(m \log m) \) time.

**Algorithm:**

- Sort edges by weight.
- Use union-find data structure to dynamically maintain connected components.

\[
\text{KRUSKAL}(V, E, w) \\
\text{SORT} \ m \text{ edges by weight so that } w(e_1) \leq w(e_2) \leq \ldots \leq w(e_m) \\
S \leftarrow \emptyset \\
\text{FOREACH } v \in V: \text{MAKESET}(v). \\
\text{FOR } i = 1 \text{ TO } m \\
\quad (u, v) \leftarrow e_i \\
\quad \text{IF FINDSET}(u) \neq \text{FINDSET}(v) \\
\quad \quad S \leftarrow S \cup \{ e_i \} \\
\quad \text{UNION}(u, v). \\
\text{RETURN } S
\]