Definition

A graph G is a finite collection of vertices V connected with edges E. Each edge in E is a pair (v_1, v_2), where v_1 and v_2 are vertices in V. Edges in a graph represent a relation between vertices like distance, time, cost or precedence. When such relation is identified with a quantity, the edge is said to be weighted and the corresponding graph is called weighted graph.

Directed Graph

A directed graph (digraph) is the graph in which for each edge in E, the ordering of pair (v_1, v_2) matters. The edge (v_1, v_2) and (v_2, v_1) is defined respectively as:

$$V_1 \rightarrow V_2$$

$$V_1 \leftarrow V_2$$

Undirected Graph

An undirected graph is the graph in which for each edge in E, the ordering of pair (v_1, v_2) does not matter. The edge (v_1, v_2) = (v_2, v_1) is defined as:

$$V_1 - V_2$$

Representation

There are two ways in which graph can be represented. A representation is chosen over the other depending on whether the graph is dense or sparse.

Adjacency Matrix

A graph can be represented using an *adjacency matrix*. For a graph G containing $n = |V|$ number of vertices, its adjacency matrix M of size $n \times n$ is defined as:

$$M_{ij} = \begin{cases} 1 & if\ (v_i, v_j) \in E \\ 0 & otherwise \end{cases}$$

For undirected graph, the matrix is symmetric since (v_1, v_2) = (v_2, v_1). This representation allows for constant time search for any edges. However, it takes $O(|V|^2) = O(n^2)$ memory space even when the graph does not have many edges.

Adjacency List

A graph is represented by $|V|$ linked lists for each vertex V storing the outgoing edges. Each edge appears in one of the linked list if the graph is directed or two linked lists if the graph is undirected. Thus

in any case, the total size of the graph is $O(|V| + |E|)$.

Decomposition

Acyclic

A cycle in a directed graph is a circular path of length at least 1 such that $v_1 = v_N$. For example, $v_2 \to v_3 \to v_4 \to v_2$ is a cycle. A cycle in an undirected graph requires edges to be distinct so that a circular path $v_1 \to v_2 \to v_1$ is not considered as a cycle, because edges (v_1, v_2) and (v_2, v_1) are the same. A directed graph is acyclic if it does not have any cycles and is referred to as a directed acyclic graph or **DAG**.

Algorithm:

Steps	Operations	Complexity
1	Store each vertices *in-degree* in an array or on each Vertex node	$O(\|E\|)$
2	Initialize a queue with all vertices of *in-degree* 0	$O(\|V\|)$
3	While queue is not empty: • Dequeue a source and delete it from the graph. • Reduce *in-degree* of all vertices adjacent to the source. Enqueue any vertices of *in-degree* 0.	$O(\|E\| + \|V\|)$

If the graph is empty at the end, then it is acyclic. Otherwise, it is cyclic. The algorithm runs in linear time $O(|E| + |V|)$.
Another way to check if a graph is acyclic is by using depth search search. The rule is that a directed graph is acyclic if and only if it has no back edges.

Topological Sort / Linearized Graph

Topology tells us how elements in a set are related to each other. In that sense, topological sorting of a directed graph $G = (V, E)$ is a linear ordering of its vertices such that for all edges $(v_1, v_2) \in E$, v_1 precedes v_2 in the ordering. It is clear that a directed graph can be linearized only if it does not contain any cycles, i.e it is a dag.

Algorithm:

Steps	Operations	Complexity
1	Store each vertices *in-degree* in an array or on each Vertex node	$O(\|E\|)$

Steps	Operations	Complexity
2	Initialize a queue with all vertices of *in-degree* 0	$O(\|V\|)$
3	While queue is not empty: • Dequeue a source, output it and delete it from the graph. • Reduce *in-degree* of all vertices adjacent to the source. Enqueue any vertices of *in-degree* 0.	$O(\|E\| + \|V\|)$

Thus, linearizability and acyclicity of a graph are infact the same property.

Strongly Connected Components

An undirected graph is said to be **connected** if there exists a path from every vertex to every other vertex. Each re-run of depth first search on an undirected graph generates a new connected components.

Connectivity property in case of a directed graph is more interesting! A directed graph is said to be **strongly connected** if there exists a path from every vertex to every other vertex. Disjoint set partitions of V thus obtained are called **strongly connected components**. It follows,

A directed graph is a dag of its strongly connected components.

Kosaraju's Algorithm:

Steps	Operations	Complexity
1	Generate reverse of a graph G (transpose)	$O(\|V\|)$
2	Run dfs on reverse graph G^R and generate a stack ordering of vertices	$O(\|V\| + \|E\|)$
3	Run dfs on graph G by processing vertices from the stack	$O(\|V\| + \|E\|)$

Kosaraju's algorithm runs in linear time but requires two graph traversal. It uses the fact that G and G^R have the same strongly connected components. **Tarjan's** and **Path-based** algorithms find strongly connected components in a single traversal.

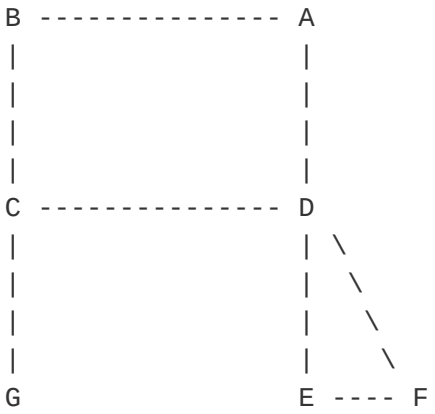
Biconnectivity

A connected undirected graph is said to be biconnected if there exists no vertex whose removal will make the remaining graph disconnected. If such vertices exist, they are known as **articulation points**.

Characteristics of a Biconnected graph:

- There exists at least two disjoint paths between any two vertices. So if a vertex is removed from one path, all the vertices will still be connected from the second path.
- There exists at least a cycle between any two vertices.
- Graph containing only two nodes and an edge connecting them is biconnected by definition.

Biconnectivity of a graph for some practical applications is very critical. For example, a network of airports should be ideally biconnected so that airplanes always have an alternate route should an airport be closed due to severe weather.



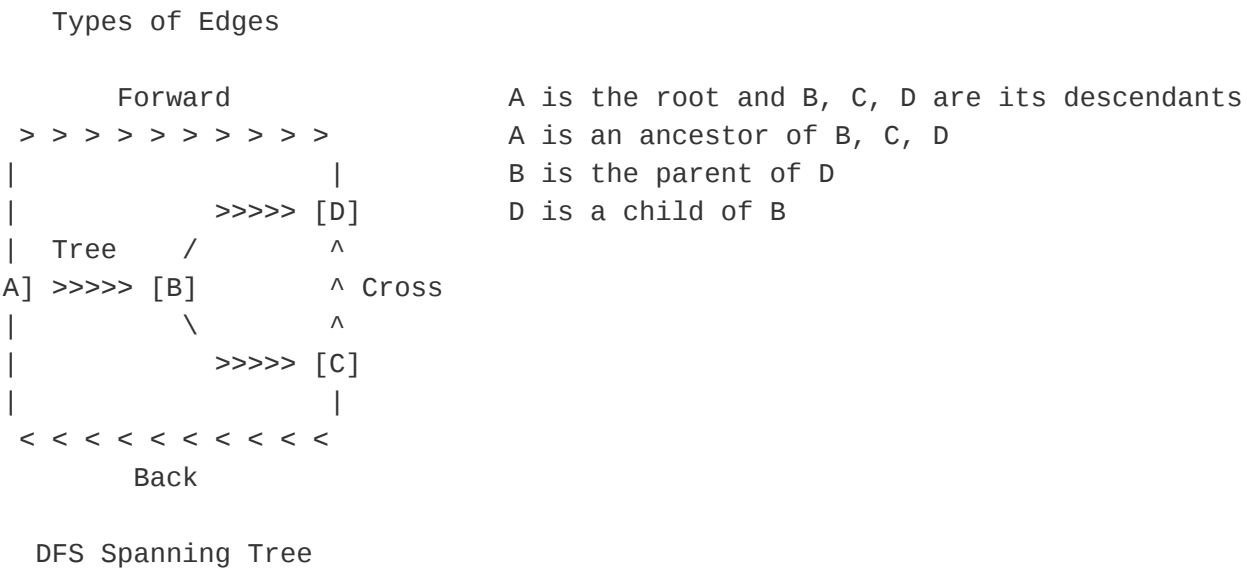
An undirected graph with two articulation points C and D

Depth first search gives us a linear time algorithm to find if there are any articulation points in the graph. DFS numbers each vertices as they appear in the search tree, call it n_v for vertex v. Based on these numbers only, each edge can be classified as a *back edge* or a *tree edge*.

- (v, w) is a back edge, if $n_v > n_w$
- (v, w) is a tree edge, if $n_v < n_w$

Furthermore, let l_v be the lowest numbered vertex that is reachable from v by taking 0 or more *tree* edges and possibly a *back* edge (in that order). Thus l_v is given by,

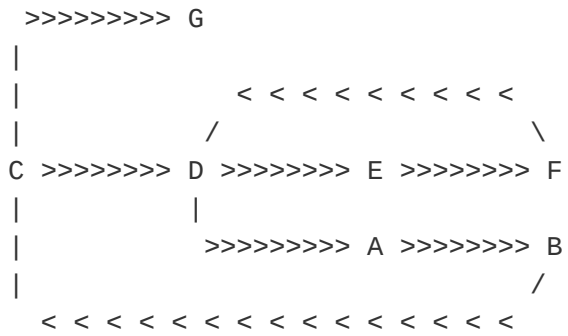
$$l_v = \min \begin{pmatrix} n_v, \\ \min_{(v,w) \in back\ edges}(n_w), \\ \min_{(v,w) \in tree\ edges}(l_w) \end{pmatrix}$$



Conditions satisfying articulation points:

- **Root** is an articulation point if it has 2 or more tree edges. This is true because if one of those edges is removed, then the two sub-trees become disconnected.

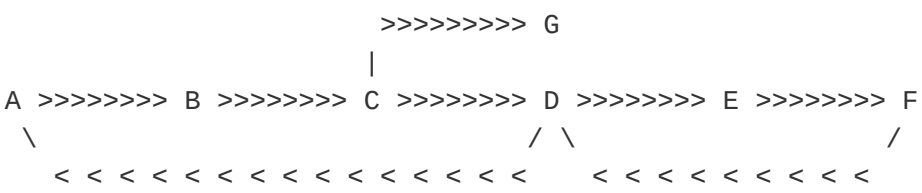
  DFS Spanning tree of the above example with C as the root.



  The root C is the articulation point as it has 2 tree edges.
  Removing C will disconnect G from rest of the graph.

- **Non-root** vertex v is an articulation point, if v has a child w such that no back edge starting in the subtree of w reaches an ancestor of v.

DFS Spanning tree of the above example with C as the root.



D is an articulation point because no back edge in the sub-tree of E reaches D's ancestor. If D is removed, then E and F will be disconnected from the rest of the graph.

Algorithm:

```
def biconnected(G, root):       # Assumes G is connected undirected graph
    number = dict()
    low = dict()
    visited = dict()
    path = dict()
    counter = 1
                                                        ----
    for each vertex v in G:                              | O(|V|)
        visited[v] = False                               |
                                                        ----
    find_articulation_points(root)                       | O(|E|)
                                                        ----
    # Check root
    tree_count = 0
    for each vertex w adjacent to root:
        if num[w] > num[root]:
            tree_count += 1
            if tree_count == 2:
                return (root + " is an articulation point")


# Recursive call to perform DFS and assign n and l values.
def find_articulation_points(v):
    visited[v]= True;
    num[v] = counter
    low[v] = num[v]
    counter += 1

    for each vertex w adjacent to v:
        if not visited[w]:
            path[w] = v
            find_articulation_points(w)

            if low[w] >= number[v]:                     # Tree Edge
                if v is not root:
                    print(v + " is an articulation point")

            low[v] = min(low[v], low[w])

        else:
            if path[v] != w:                            # Back Edge
                low[v] = min(low[v], number[w])
```

The algorithm runs in linear time $O(|V| + |E|)$.

Paths

Path in a graph is a sequence of vertices $v_1, v_2, ..., v_N$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < N$. Let $c_{i,j}$ be the cost to traverse the edge (v_i, v_j). Then, the length of the path $v_1, v_2, ..., v_N$ is given as,

$$\sum_{i=1}^{N-1} c_{i,i+1}$$

If the graph is undirected, then $c_{i,j} = 1,\ \forall\ (i, j)$ and the path length is $N - 1$. A shortest path in a graph thus refers to the path with minimum cost of traversal. The question we would like to answer is:

Given a graph $G = (V, E)$ and a source vertex s, find shortest paths from s to all vertices in G.

Single-Source Shortest Path for Unweighted Graph

The shortest path for unweighted graph can be computed using simple breadth first search exploration. The algorithm runs in linear time $O(|E| + |V|)$.

BFS Algorithm:

```
def shortest_path(G, s):                                          ----
    distance = dict(((vertex, float("inf")) for vertex in G))       | O(|V|)
    path = dict()                                                   |
                                                                   ----
    queue = Queue()
    distance[s] = 0
    queue.enqueue(s)

                                                                  -----
    while queue is not empty:                                       |
        v = queue.dequeue()                                         |
                                                                    |
        for each vertex w adjacent to v:                            |
            if distance[w] is infinity:                             | O(|E|)
              distance[w] = distance[v] + 1                         |
              path[w] = v                                           |
              queue.enqueue(w)                                      |
                                                                  -----
```

Single-Source Shortest Path for Weighted Digraph with non-negative edges

Dijkstra algorithm is a greedy algorithm to compute the shortest path for a weighted digraph. At each step, the algorithm chooses the closest node and relaxes the corresponding edge irrespective of its future implications.

Dijkstra algorithm always generates shortest path if edge weights are non-negative.

We can prove above statement using induction. Consider a partition S of nodes, where all nodes have shortest path from the source.

- **Base Case:** If the partition only contains source, shortest path is simply 0.
- **Inductive Hypothesis:** Suppose the partition contains $k-1$ nodes and all have shortest path from the source.
- **Inductive Step:** The path length to the next closest node M is equal to the edge cost plus the distance of the parent node from the source. Since distance of the parent node from the source is the shortest path (from the hypothesis), this path to M must be the shortest. Thus, k nodes have the shortest path from the source.

The performance of Dijkstra algorithm depends heavily on the priority queue implementations used.

Priority Queue	$deletemin$	$insert$	**Overall complexity**
Array	$O(\|V\|)$	$O(1)$	$O(\|V^2\|)$
Binary Heap	$O(log\|V\|)$	$O(log\|V\|)$	$O((\|V\|+\|E\|)log\|V\|)$
d-array Heap	$O(\frac{dlog\|V\|}{logd})$	$O(\frac{log\|V\|}{logd})$	$O((\|V\|\times d+\|E\|)\frac{log\|V\|}{logd})$
Fibonacci Heap	$O(log\|V\|)$	$O(1)$ (amortized)	$O(\|V\|log\|V\|+\|E\|)$

Dijkstra Algorithm:

```
def shortest_path(G, s):
    distance = dict()
    path = dict()
    known = dict()
    priorityqueue = BinaryHeap()

    for each vertex in G:                                ----
        distance[vertex] = float("inf")                      |
        path[vertex] = None                                  | O(|V|)
        known[vertex] = False                                |
                                                         ----
    distance[s] = 0                                      ----
    priorityqueue.buildheap(distance, vertices)              | O(|V|)
                                                         ----
    while priorityqueue is not empty:                    ----
        _ , v = priorityqueue.deletemin()                    |
        if known[v]: continue                                |
                                                             |
        known[v] = True                                      |
        for each vertex w adjacent to v:                     | O((|E|+|V|)log|V|)
            if distance[v] + cost(v,w) < distance[w]:        |
                distance[w] = distance[v] + cost(v,w)        |
                path[w] = v                                  |
                priorityqueue.insert((distance[w], w))       |
                                                         ----
```

Single-Source Shortest Path for Weighted Digraph with negative edges

Dijkstra algorithm does not work with negative edges. It always gives the shortest path from a source s to a node v in case of non-negative edges because the shortest path must exclusively pass through all nodes that are closer than v. This does not hold when there are negative edges.

Thus, instead of *updating* or *relaxing* the nearest edge, all edges are updated $|V| - 1$ times. $|V| - 1$ is the maximum number of edges possible in a shortest path when the graph contains V vertices. This algorithm is known as **Bellman-Ford algorithm** and has a complexity. $O(|V| \times |E|)$. It also checks if there exists a negative cycle, under which case the shortest path is not defined.

Bellman-Ford Algorithm:

```
def shortest_path(G, s):                                    ----
   distance = dict(((vertex, float("inf")) for vertex in G))   | O(|V|)
   path = dict()                                               |
                                                            ----
    distance[s] = 0
                                                            ----
    for _ in range(|V|-1):                                      |
        for each edge (v,w) in G:                               |
            if distance[v] + cost(v,w) < distance[w]:           | O(|V|.|E|)
                distance[w] = distance[v] + cost(v,w)           |
                path[w] = v                                     |
                                                            ----
    for each edge (v,w) in G:
        if distance[v] + cost(v,w) < distance[w]:
            raise NegativeCycleError
```

Instead of blindly *relaxing* all vertices, in **SPFA** a queue is maintained and a vertex is added in the queue only once it is *relaxed*. Even though it improves average running time, the worst-case run-time is still $O(|V| \times |E|)$. This is because in the worst-case, each vertex can be dequeued $|V|$ number of times.

Shortest Path Faster Algorithm (SPFA):

```
def shortest_path(G, s):
    distance = dict()
    path = dict()
    in_queue = dict()
    queue = Queue()

    for each vertex in G:                                  ----
        distance[vertex] = float("inf")                        |
        path[vertex] = None                                    | O(|V|)
        in_queue[vertex] = False                               |
                                                           ----
    distance[s] = 0
    queue.enqueue(s)
                                                           ----
    while queue is not empty:                                  |
        v = queue.dequeue()                                    |
        for each vertex w adjacent to v:                       |
            if distance[v] + cost(v,w) < distance[w]:          |
                distance[w] = distance[v] + cost(v,w)          | O(|V|.|E|)
                path[w] = v                                    |
                if not in_queue[w]:                            |
                    queue.enqueue(w)                           |
                    in_queue[w] = True                         |
                                                           ----
```

DFS is another algorithm based on depth first search and closely resembles SPFA. Vertices are *relaxed* in a depth first search manner. Both **SPFA** and **DFS** can be terminated in case of negative cycle detection. A negative cycle is detected when any of the vertices are *dequeued* or *popped* $|V| + 1$ number of times.

DFS:

```
def shortest_path(G, s):
    distance = dict()
    path = dict()
    in_stack = dict()
    stack = Stack()

    for each vertex in G:                                  ----
        distance[vertex] = float("inf")                        |
        path[vertex] = None                                    | O(|V|)
        in_stack[vertex] = False                               |
                                                           ----
    distance[s] = 0
    stack.push(s)
                                                           ----
    while queue is not empty:                                  |
        v = stack.pop()                                        |
        for each vertex w adjacent to v:                       |
            if distance[v] + cost(v,w) < distance[w]:          |
                distance[w] = distance[v] + cost(v,w)          | O(|V|.|E|)
                path[w] = v                                    |
                if not in_stack[w]:                            |
                    stack.push(w)                              |
                    in_stack[w] = True                         |
                                                           ----
```

Single-Source Shortest Path for Directed Acyclic Graph (DAG)

If a graph is dag, then shortest path from a source can be computed in linear time. This is because vertices are arranged in a linearized order in every part of the dag. Thus, it is sufficient to linearize graph and then *relax* edges by choosing vertices in order. The algorithm completes in one pass with complexity $O(|V| + |E|)$!

Algorithm:

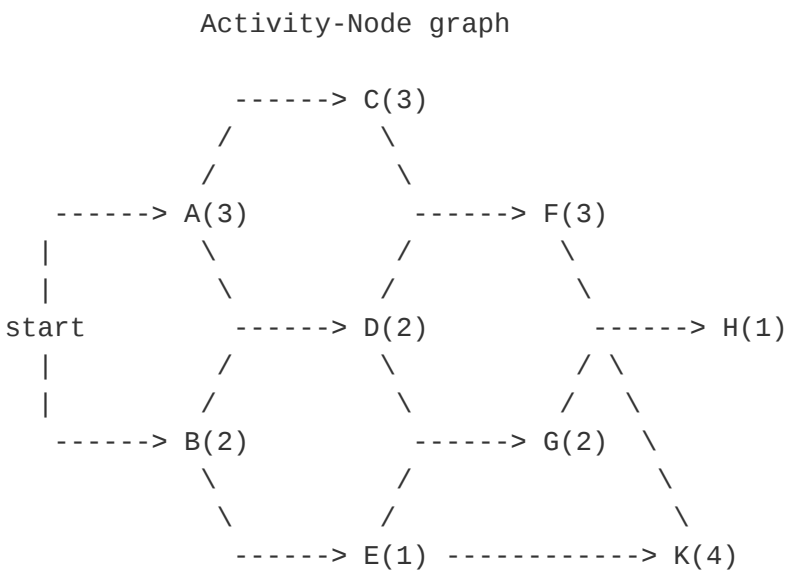
```
def shortest_path(G, s):
    distance = dict()
    path = dict()

    for each vertex in G:                                   ----
        distance[vertex] = float("inf")                         |
        path[vertex] = None                                     | O(|V|)
                                                            ----
    distance[s] = 0
    linearize(G)
                                                            ----
    for each vertex v in linearized order:                  |
        for each vertex w adjacent to v:                    |
            if distance[v] + cost(v,w) < distance[w]:       | O(|E|)
                distance[w] = distance[v] + cost(v,w)       |
                path[w] = v                                 |
                                                            ----
```

Critical Path in Directed Acyclic Graph

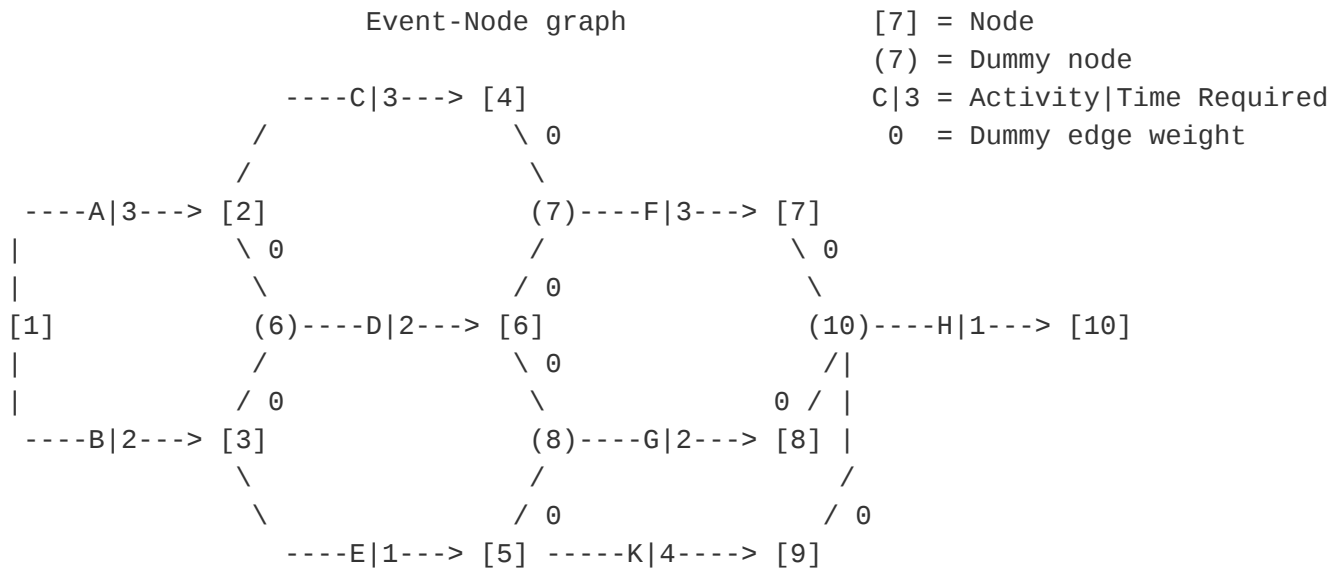
Acyclic graphs are very useful to model temporal, hierarchical or downhill problems. As an example, consider you are working on a project that has many sub-tasks and you would like to answer following questions:

- What is the earliest time of completion of the project?
- How much can a task be delayed without affecting the earliest time of completion of the project?
- Which tasks are critical and cannot not be delayed in order to complete project on time?

Consider an activity-node graph. Each node represents the activity that must be performed along with the time it takes to complete the activity. Each edge (v, w) represents a precedence relation in which activity v must be completed before activity w.



In order to answer above questions, we first transform the activity-node graph into an event-node graph. Each node represents a completion of activity **and** all its dependencies. There might be a need to add dummy nodes and edges. In the event-node graph shown below, (6) is a dummy node (shown in parentheses) that indicates completion of **both** A and B tasks.



Earliest completion time (EC) corresponds to the longest path in the event-node graph. It is not shortest path, as it might seem, because of the dependencies of each activity. Furthermore, it is important to note that longest path, like smallest path, also suffers from the existence of **positive cycles** in presence of positive edges. However, since event-node graph is always acyclic, positive cycles do not exist. Earliest completion time is computed for each vertices in topological order. It is computed by the following recursive relation:

$$EC_1 = 0$$

$$EC_w = \max_{(v,w)\in E}(EC_v + c_{v,w})$$

Latest completion time (LC) is the time by which each activity can be completed without delaying the final completion time. It is computed for each vertices in reverse topological order. It is given by the following recursive relation:

$$LC_n = EC_n$$

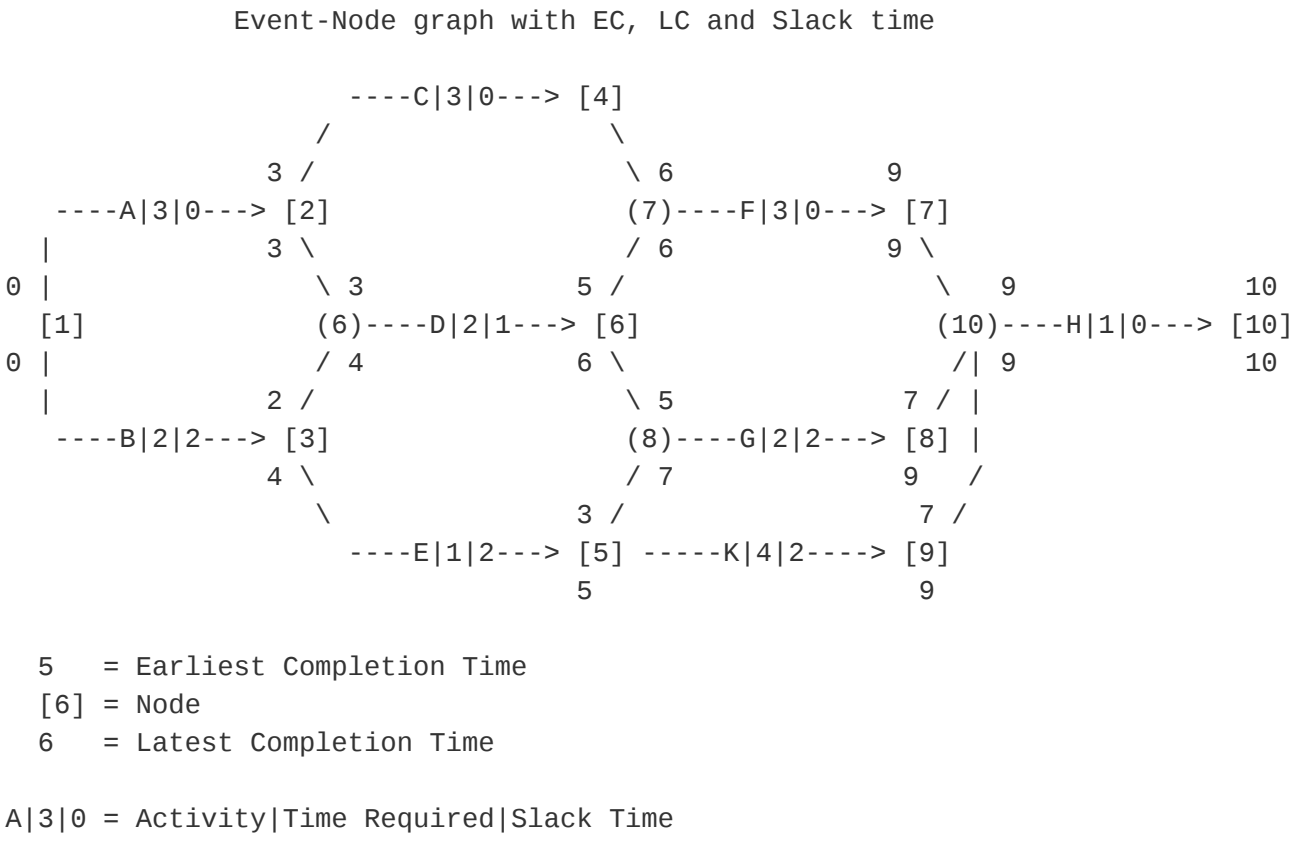
$$LC_v = \min_{(v,w)\in E}(LC_w - c_{v,w})$$

Slack time for each edge in the event-node graph is the time by which the corresponding activity can be delayed without delaying the final completion time. It is given by the following relation:

$$Slack_{(v,w)} = LC_w - EC_v - c_{v,w}$$

These values can be calculated in linear time by maintaining for each node a list of its precedent and adjacent nodes.

Finally, the **critical path** is the path with all edges having 0 slack time. The activities in this path are critical and will need to be finished on time. There is always at least one critical path in the event-node graph. The illustration below shows EC, LC and $Slack$ time for the example chosen.



As shown above, task D can be finished earliest in 5 unit time and latest in 6 unit time. It can be delayed 1 unit time without delaying final completion time of 10 unit. Similarly task B can be delayed for 2 unit time. The path $A \rightarrow C \rightarrow F \rightarrow H$ is critical and thus tasks A, C, F, H must be completed on time.

Flows

Graphs can be used to study flow of water through network of pipes or flow of traffic on interconnected streets. Consider a graph $G = (V, E)$ in which each edge is associated with a capacity $c_{v,w}$ that represents the amount of water or the amount of traffic. Flow is analyzed between a source vertex s and a sink vertex t. Any vertices between them must conserve flow i.e. *incoming flow* = *outgoing flow*. Through any edge (v, w) at most $c_{v.w}$ flow may pass. The resulting (G, c, s, t) is called the **flow network**.

Maximum Flow

Max-Flow Min-Cut Theorem

Suppose a graph is cut into two partitions such that source s and sink t are in different partitions. Then, the maximum flow from source to sink is bounded by the total capacities of all edges (v, w) where v is in s's partition and w is in t's partition. Of all the possible cuts, the cut with minimum capacity is exactly equal to the maximum flow.

A simple algorithm for finding maximum flow in a flow network is based on the idea of repeatedly adding an admissible flow path generated from the residual graph onto a flow graph until no more such paths exist. Formalizing the concept, let

- $G = (V, E), c, s, t$ be the **flow network**.
- G_f be the flow graph, initialized such that $\forall\ (v, w) \in E,\ c_{v,w} = 0$.
- $f_{v,w}$ be the flow between vertices v and w in G_f.
- G_r be the residual graph, initialized to G.

The algorithm proceeds by finding an **augmenting path** p from s to t in G_r, adding the smallest flow capacity in p to corresponding edges in G_f and updating the G_r. The process is repeated until no such augmenting path is found in G_r, in which case the algorithm terminates. This method is popularly known as **Ford-Fulkerson method** of finding the maximum flow. (*method* since approach of finding the augmenting path is not defined)

The residual graph G_r is updated in two steps:

- Decrease capacities of edges in the augmenting path: $c_{v,w} = c_{v,w} - f_{v,w},\ \forall\ (v, w) \in p$
- Add corresponding reverse edges: $add\_edge((w, v), f_{v,w}),\ \forall\ (v, w) \in p$

There are multiple approaches of generating augmenting path from the residual graph. Two of such approaches are discussed below.

Edmonds-Karp Algorithm:

Edmonds-Karp algorithm is an implementation of Ford-Fulkerson method in which the shortest path generated from BFS is used as the augmenting path. Thus generated shortest augmenting path is also a *monotonically increasing* function. This means the length of a shortest path found in the $i+1$th step is *never* less than the length of a shortest path found in the ith step.

Edmonds-Karp algorithm guarantees termination and runs in $O(|V| \times |E|^2)$
Each augmentation requires generating path through BFS search that takes $O(|E|)$. Furthermore, each augmenting path saturates at least 1 edge. Suppose (v, w) is the saturated edge and the vertex v is at a distance d_v from the source s. After augmentation, edge (v, w) is removed (since its saturated) and its reverse edge (w, v) is added to G_r. The edge (v, w) can only reappear in G_r if the future augmenting path contains the edge (w, v) and it is saturated. When it reappears, d_v must be $d_w + 1$. Since d_w was $d_v + 1$, it is clear that d_v has increased by 2. Since v's distance from the source can be at most $|V|$, it can only reappear $|V|/2$ number of times. Thus each $O(|E|)$ edges can reappear $O(|V|)$ number of times and the total number of augmentation is $O(|E| \times |V|)$. It follows, the total complexity of Edmonds-Karp algorithm is $O(|E| \times (|E| \times |V|)) = O(|V| \times |E|^2)$.

Algorithm:

```
- Defined residual graph G_r, initialized to G
- Initialize maximum flow to 0.
- While there is augmenting path p from s to t in G_r:
      Find the smallest flow capacity in the path.
      Increment maximum flow by the smallest flow capacity.
      Decrease capacities of edges in the augmenting path in G_r.
      Add corresponding reverse edges in G_r.
- Return maximum flow.
```

Pseudocode:

```
def maximum_flow(G, s, t):
    max_flow = 0
    G_r = residual(G)
    path = BFS(G_r, s, t)

    while path is not None:
        flow = min(capacities in path)
        max_flow = max_flow + flow

        for each edge (v,w) in path:
            c_vw = c_vw - flow
            G_r.add_edge((w,v), flow)

    return max_flow
```

Maximum Capacity Path Algorithm:

The algorithm is also known as **Gradient modification of the Ford-Fulkerson method**. Instead of choosing shortest path from BFS, the algorithm chooses path with maximum capacity using Dijkstra algorithm. A downside of this algorithm is that it does not guarantee termination for irrational capacities. A good example is demonstrated in the Wikipedia. However, if using integral capacities, it always terminates and gives the maximum flow value.

Maximum Capacity Path Algorithm runs in $O(|E|^2 \times log|V| \times log(c_{max}))$
It can be shown that if c_{max} is the maximum flow capacity in the graph, then $O(|E| \times log(c_{max}))$ number of augmentation is sufficient to find the maximum flow. Since Dijkstra algorithm (using binary heap) runs in $O(|E| \times log|V|)$ for finding each augmenting path, the total complexity is $O(|E|^2 \times log|V| \times log(c_{max}))$.

Algorithm:

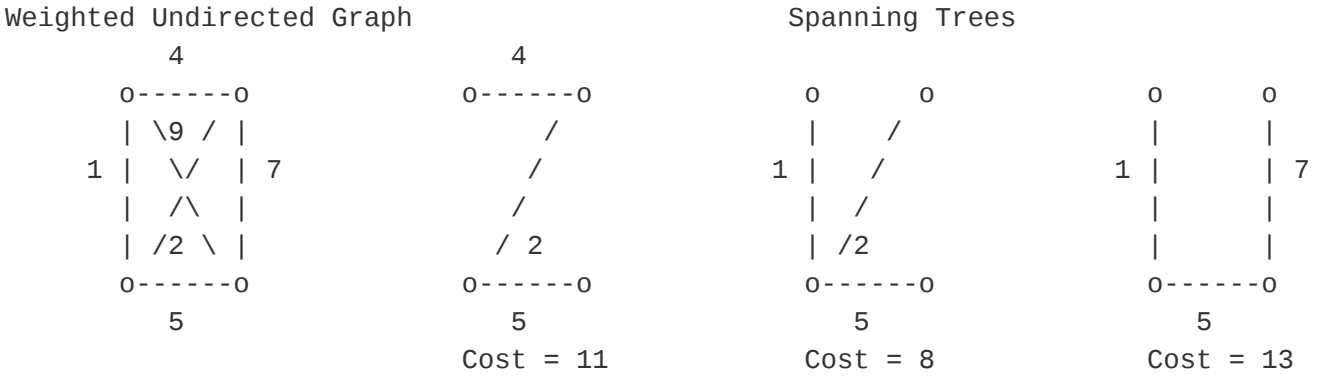
```
- Defined residual graph G_r, initialized to G
- Initialize maximum flow to 0
- While there is augmenting path p from s to t in G_r:
      Find the smallest flow capacity in the path
      Increment maximum flow by the smallest flow capacity
      Decrease capacities of edges in the augmenting path in G_r
      Add corresponding reverse edges in G_r
- Return maximum flow.
```

Pseudocode:

```
def maximum_flow(G, s, t):
    max_flow = 0
    G_r = residual(G)
    path = Dijkstra(G_r, s, t)

    while path is not None:
        flow = min(capacities in path)
        max_flow = max_flow + flow

        for each edge (v,w) in path:
            c_vw = c_vw - flow
            G_r.add_edge((w,v), flow)

    return max_flow
```

Spanning Tree

A spanning tree of graph $G = (V, E)$ is a sub-graph G containing all vertices V and a sub-set of edges E $(|V| - 1)$ that touches all vertices without any cycles. It is a sparse subgraph that uncovers many properties of the underlying graph.



Minimum Spanning Tree is the spanning tree with the least total edge cost. In above example, the second tree with cost 8 is the MST. **Minimum Spanning Forest** is the union of minimum spanning trees for a graph with multiple connected components. If all edges in the graph are distinct, then there will be only one, *unique* minimum spanning tree.
MST allows us to answer many important practical questions like:

- What is the minimum length of gas pipelines between connected cities?
- Find cheapest way to wire a house (with minimum cable).
- Find a way to connect various routers on a network that minimizes total delay.

Thus, many algorithms have been designed to efficiently find MST's. Two of such algorithms are discussed below.

Prim Algorithm

Prim Algorithm is a greedy algorithm that proceeds pretty much like Dijkstra algorithm. The distinction is in how *distance* is defined and updated. *distance* of a vertex is not its distance from the source like in Dijkstra, but rather its distance from a *known* vertex. Every time a smaller distance from a known vertex is found, *distance* is updated for that vertex. The complexity analysis is identical to that of the Dijkstra algorithm. It runs in $O((|E| + |V|)log|V|)$ when binary heap is used.

Algorithm:

```
- Initialize distance of each node to "inf" and mark it unknown
- Initialize distance of one selected node s to 0, with distance[s] = 0
- While there are unknown nodes left in the graph:
      Select the unknown node v with the lowest cost
      Mark v as known
      For each unknown node w adjacent to v:
          If cost of (v, w) < v's cost:
              v's cost = cost of (v, w)
              distance[w] = v
```

Pseudocode:

```
    def mst(G,root):
        distance = dict()
        path = dict()
        known = dict()
        priorityqueue = BinaryHeap()

        for each vertex in G:                                ----
            distance[vertex] = float("inf")                      |
            path[vertex] = None                                  | O(|V|)
            known[vertex] = False                                |
                                                               ----
        distance[root] = 0                                     ----
        priorityqueue = buildheap(distance, vertices)            | O(|V|)
                                                               ----
        while priorityqueue is not empty:                      ----
            _ , v = priorityqueue.deletemin()                    |
            if known[v]: continue                                |
                                                                 |
            known[v] = True                                      |
            for each vertex w adjacent to v:                     | O((|E|+|V|)log|V|)
                if cost(v,w) < distance[w]:                      |
                    distance[w] = cost(v,w)                      |
                    path[w] = v                                  |
                    priorityqueue.insert((distance[w], w))       |
                                                               ----
```

Kruskal Algorithm

Kruskal algorithm is also a greedy algorithm that selects edges in the order of smallest weights and accepts the edge if it does not create a cycle. The algorithm initializes a forest containing single node trees. When an edge is accepted, the corresponding vertices are merged to form a tree. After selecting $|V| - 1$ edges in order, a single tree containing all vertices is obtained.

In order to efficiently merge trees and detect cycles, **Disjoint Set** data structure is used. The equivalence relation that guides elements in different disjoint sets is "connectivity". Two vertices v and w belong to the same disjoint set if there exists an edge (v, w) or (w, v) to connect them.

A cycle is generated in a tree T, when an edge (v, w) such that $v, w \in T$ is added. Thus, checking if an edge generats a cycle in a tree is merely checking if the corresponding two vertices are in the same disjoint set.

The Kruskal algorithm runs in $O(|E| \times log|E|)$. This is because, in the worst case, in order to select $|V| - 1$ edges, the algorithm might have to *deletemin()* $|E|$ number of edges each taking $O(log|E|)$ time.

Algorithm:

```
- Put all the vertices into single node trees by themselves
- Put all the edges in a priority queue with key = edge cost
- Repeat until |V|-1 edges have been accepted:
      Extract cheapest edge
      If it forms a cycle:
          ignore it
      else:
          accept the edge – it will join two existing trees and yield a larger tree
- Return the accepted edges (they form the spanning tree)
```

Pseudocode:

```
def minimum_spanning_tree(G):
    mst = []                                              ----
    dsets = DisjointSets(all vertices in G)                   | O(|V|)
    priorityqueue = BinaryHeap()                          ----
                                                          ----
    priorityqueue.buildheap(all edges (c,v,w) in G)           | O(|E|)
                                                          ----
    while len(mst) != |V|-1:                              ----
        c,v,w = priorityqueue.deletemin()                     |
        vset = dsets.find(v)                                  |
        wset = dsets.find(w)                                  |
        if vset != wset:                                      | O(|E|.log|E|)
            dsets.union(vset, wset)                           |
            mst.append((v,w))                                 |
                                                          ----
    return mst
```

Tours and Circuits

Euler Tour / Euler Circuit

Euler tour is a path in a graph that passes through each edge only once. If the initial and final vertex is the same, the path is called **Euler circuit**. A graph containing an Euler circuit or Euler cycle is called **Eulerian Graph**. Euler found out that for such path to exist in a graph, it must be connected and all its vertices must have even degrees. This is true because one edge is required to enter a vertex and another one to exit.

Furthermore,

- If one vertex has odd degree, then Euler tour is possible only if the path starts from the odd degree vertex.
- If two vertices have odd degrees, then Euler tour is possible only if the path starts from one and ends at the other odd degree vertex.
- If more than two vertices have odd degrees, then Euler tour is not possible.

An undirected graph can be checked for being Eulerian in linear time by simply checking if all vertices has even **degrees**. Similarly, for directed graph it is sufficient to check if all vertices have equal **in-degree** and **out-degree**. If a graph is known to be Eulerian, its Euler circuit can be found using the algorithm described below.

Hierholzer Algorithms:

```
- Do a depth-first search (DFS) from a vertex until you are back at this vertex.
- Pick a vertex on this path with an unused edge and repeat 1.
- Splice all these paths into an Euler circuit.
```

Pseudocode:

```
def euler_circuit(G, source):       # Assumes G is Eulerian
    stack = Stack()
    circuit = []
    G_copy = G.deepcopy()

    circuit.append(source)
    while True:                                          ----
        if source.out_degree > 0:                            |
            stack.push(source)                               |
            G_copy.del_edge((source, source.adjacent())      |
            source = source.adjacent()                       |
        else:                                                |
            source = stack.pop()                             | O(|E|)
            circuit.append(source)                           |
                                                             |
        if stack is empty:                                   |
            break                                            |
                                                           ----
    return reverse(circuit)
```

Hamiltonian Tour / Hamiltonian Circuit

Hamiltonian Tour is a path in a graph that passes through each vertex only once. If the initial and final vertex is the same, the path is called **Hamiltonian circuit**. A graph that contains a Hamiltonian path is called a **Traceable graph**. A graph that contains Hamiltonian circuit/cycle is called **Hamiltonian graph**. All Hamiltonian graphs are biconnected.

There are no known efficient algorithm to find such a path. A brute force algorithm, that finds all possible paths and choose one where each vertex is visited only once, can be used but runs in exponential time. If B is the average size of any vertex's adjacency list (*branching factor* in terms of DFS tree), then it would take $B^{|V|}$ time.

```
Graph A is Hamiltonian but not Eulerian.
Graph B is both Hamiltonian and Eulerian.
```