Summary of Lectures

ITC2, Spring 2022

by:

Padmini Mukkamala

Budapest University of Technology and Economics

Last updated: June 16, 2022

The pseudocodes that appear here have been adapted from the Hungarian Course Notes.

Note: For the final exam, you have to know the proofs of theorems in green and not for the ones in yellow.

Contents

Lecture 1

Permutations, Combinations, Binomial Theorem.

Permutations, Combinations

Picking k out of n items.

Experiment	With replacement	Without replacement
Ordered	n^k	$n(n-1)...(n-k+1)$
Unordered	$\binom{n+k-1}{k}$	$\binom{n}{k}$

Binomial Theorem

Relations of Binomial coefficients $\binom{n}{k} = \binom{n}{n-k}$, $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.
Proof sketch: $\binom{n}{k} = \frac{n!}{k!(n-k)!} = \binom{n}{n-k}$. But we can also see this with a combinatorial argument. $\binom{n}{k}$ is the number of ways to pick k out of n distinct items (no repetition, order doesn't matter). But we could also first choose the $n-k$ objects to discard and this gives us another way to count the number of ways to pick k out of n items. But the number of ways to pick $n-k$ out of n objects is $\binom{n}{n-k}$.

For the second equality, combinatorial proof: To count the number of ways to pick k out of n distinct items, we can first count all the subsets which contain the last (n-th) item and then all the subsets which do not contain n.

Binomial Theorem: $(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$ and its proof.
Proof sketch: There are n factors $x+y$, so to get a term in the expansion, we must pick one literal (x or y) from each factor. If the term is $x^k y^{n-k}$, then we must pick x from exactly k factors and this can be done in $\binom{n}{k}$ ways.

This result can also be proved using induction on n. Base case $n = 0$.

Pascal's triangle: a triangular array of binomial coefficients where genarating the next row uses the equality $\binom{n}{k} + \binom{n}{k-1} = \binom{n+1}{k}$.

Lecture 2

Graphs, isomorphism, connected graphs, components.

Graphs: Basic Definitions

Graph: a tuple $G = (V, E)$ where V (also written as $V(G)$) is the set of vertices and E (also written as $E(G)$), is a multiset of pairs of vertices.

Multiple edges: if a pair (u, v) of vertices occurs more than once in $E(G)$, then its called a multiple edge.
Loop: A pair of the form $(u, u) \in E(G)$ is called a loop.
Simple graph: if G does not contain any multiple edges or loops.
Complete graph: a simple graph where $\forall u, v \in V(G), u \neq v$, $(u, v) \in E(G)$.

Bipartite graph: if there is a partition of the vertex set into two parts, $V = V_1 \cup V_2, V_1 \cap V_2 = \phi$ and if $(u, v) \in E(G)$, then u, v belong to different partitions.
Degree: $d(v)$ or the degree of a vertex v is the number of edges incident on it.
Degree sequence: The list of degrees of the vertices of the graph written in increasing order is called its degree sequence.

Theorem (handshake theorem): $\sum_v d(v) = 2|E(G)|$.
Proof sketch: when counting the sum of degrees, every edge is counted precisely twice, once for each vertex incident on it. We assume that loops add 2 to the degree of the vertices they are incident to.

Subgraph: A graph $G' = (V', E')$ is said to be a subgraph of $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$.
Induced subgraph: A graph $G' = (V', E')$ is said to be an induced subgraph of $G = (V, E)$, if $V' \subseteq V$ and $E' \subseteq E$, and $\forall u, v \in V'$, $(u, v) \in E'$ if and only if $(u, v) \in E$.
Complement: The complement of a graph $G = (V, E)$, denoted by $\overline{G}$, is a graph such that $V(\overline{G}) = V$ and $(u, v) \in E(\overline{G})$ if and only if $(u, v) \notin E$.

Isomorphism

Isomorphism: Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be isomorphic if there is a bijection (invertible 1-1 function) $f : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$.
Note: Isomorphism preserves degrees, degree sequence, paths, cycles of the graphs.

Paths and connectivity

Walk: A walk in a graph G is a sequence $u_0, e_1, u_1, e_2, u_2, ..., e_k, u_k$ such that $e_i = (u_{i-1}, u_i) \in E(G)$. The length of the walk is k, the number of edges in it.
Trail: is a walk with no repeating edges $e_i \neq e_j$ for $i \neq j$.
Closed walk: walk with $u_0 = u_k$.
Closed trail or circuit: trail with $u_0 = u_k$.
Path: Walk with no repeating vertex, that is, $u_i \neq u_j$ for $i \neq j$.
Cycle: Closed trail with no repeating vertex, that is, $u_i \neq u_j$ for $i \neq j$.

Theorem: if $\exists$ a uv-walk, then $\exists$ a uv-path.
Proof sketch: if no vertex is repeated, then its a path. If a vertex, say u_i is repeated, delete the walk between the first and last occurances of u_i. Repeat until no vertex repeats.

Connected graphs: Graph is said to be connected if there is a path between every pair of vertices.
Components: The components of a graph are its maximal connected subgraphs.

Lecture 3

Trees, spanning tree, BFS.

Trees

Definition: Tree is a connected acyclic graph.

Theorem: the following are equivalent:

1. G is a tree.

2. $\forall u, v, u \neq v$, there is a unique uv-path.
3. G is maximally acyclic
4. G is minimally connected

(in the last two, maximal and minimal are with respect to addition or deletion of edges and not vertices).
Proof sketch: We will show $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 4 \Rightarrow 1$.
$1 \Rightarrow 2$: For contradiction, let there be two distinct paths between u and v. If the two paths have no vertices in common, then that gives us a cycle. If they do have common vertices, pick the first common vertex u' of the two paths. The two paths will give us a cycle between u and u'.
$2 \Rightarrow 3$: A cycle gives us two paths between the vertices of the cycle, so assuming 2, it is clear that G is acyclic. Further let $(u, v) \notin E(G)$. There is a unique u, v-path in G, so adding the edge (u, v) will induce a cycle, so G is maximally acyclic.
$3 \Rightarrow 4$: G must be connected, otherwise we could add an edge between components and not induce a cycle. On the other hand, let $(u, v) \in E(G)$ such that removing this edge does not disconnect the graph. But then there is a u, v-path that does not use the edge (u, v). Then this path with the edge gives us a cycle in G, which is a contradiction since G is acyclic.
$4 \Rightarrow 1$: If G has a cycle, then removing any one edge of the cycle will not disconnect the graph. So G must be acyclic, and hence a tree.

Theorem: every tree with at least 2 vertices has at least 2 leaves.
Proof sketch: Consider the longest path in G. The first and last vertex of the path must be leaves as G has no cycles and this is the longest path.

Theorem: if v is a leaf of tree G, then $G \setminus v$ is also a tree.
Proof sketch: Deleting a leaf cannot disconnect the graph or induce cycles.

Theorem: If G is a tree on n vertices, then it has $n - 1$ edges.
Proof sketch: We use induction on n. Base case, $n = 1$ is an isolated vertex. For any larger $n > 1$, we delete a leaf of the tree and use the previous theorem to realize that the new graph is a tree with $n - 1$ vertices. The result follows by using the Induction Hypothesis.

Corollary: Any connected graph G with $n - 1$ edges is a tree.
Proof sketch: If the graph G has a cycle, removing one edge from the cycle will not disconnect the graph. We repeat this process until G is acyclic. If G was not acyclic to start with, we will get a tree on n vertices with strictly fewer than $n - 1$ edges, a contradiction.

Spanning tree: Given any graph G, a spanning tree T is a subgraph of G such that $V(T) = V(G)$.

Theorem: a graph G has a spanning tree if and only if it is connected.
Proof sketch: If a graph has a spanning tree, then it is clearly connected, because a tree is connected and the spanning tree is a subgraph of G. On the other hand, if G is connected, we repeat the algorithm above of picking a cycle in G and removing an edge in it, so that we always get a subgraph of G that is still connected, but we keep reducing the cycles in it. This process is finite because the graph has finitely many edges, and when this process ends, we will be left with a acyclic connected subgraph on $V(G)$, thereby a spanning tree.

Breadth First Search

BFS: Input: G and s (starting vertex). Important lists in the table:

- i: iteration
- $b(i)$: ith vertex seen
- $d(v)$: distance from s ($d(s) = 0$ and initially $d(v) = \infty, \forall v \neq s$) (Note: for a disconnected graph, this is set to 0 for the new vertex we could not reach from s and from which we restart the BFS).
- $p(v)$: parent of v (this is always empty for s and initially set empty for every other vertex. Note that if G is disconnected, there will be several vertices without a parent, exactly one for each component).

Input: $G = (V, E)$ and a vertex $s \in V$.

```
1.  $j \leftarrow 1$; $k \leftarrow 1$; $b(1) \leftarrow s$
2.  $d(s) \leftarrow 0$; for all other vertices $v \in V, v \neq s$, $d(v) \leftarrow *$
3.  loop
4.      if $b(k)$ has a neighbor for which $d(v) = *$, then:
5.          $j \leftarrow j + 1$
6.          $b(j) \leftarrow v$
7.          $d(v) \leftarrow d(b(k)) + 1$
8.          $p(v) \leftarrow b(k)$
9.      otherwise:
10.         if $k = j$, then:
11.             stop
12.         otherwise:
13.             $k \leftarrow k + 1$
14. end loop
```

Note: j denotes the index of the vertex last discovered while k denotes the index of the vertex being currently explored or expanded.

The BFS algorithm takes at most $c \cdot (n + m)$ steps, where $n = |V(G)|$ and $m = |E(G)$.

Claim: For any edge $uv \in E(G)$, $|d(u) - d(v)| \leq 1$.
Proof sketch: without loss of genarality, let $d(u) \leq d(v)$. If v has not been explored by the time we explore u, then the algorithm will list v when it explores u and then $d(v) = d(u) + 1$. If v has been explored, then the claim is trivially true.

Theorem: the distance given by the BFS between s and any vertex v is the shortest distance between s and v, that is, there is no shorter path between them.
Proof sketch: using the claim above, the distance $d(u)$ of intermediate vertices on any path from s to v can at most be 1 more than the previous vertex. So length of the path is $\geq d(v)$.

Not covered in class but in syllabus (discussed in recitations): can use BFS to check if graph is disconnected. And if it is, then we restart the BFS from a new unexplored vertex v (with $d(v) = 0$ and $p(v)$ empty as for s) to get the BFS forest. The algorithm works just the same for directed graphs. The BFS also gives shortest distance in the directed version, only one has to be careful to note that $d(u,v)$ is not necessarily equal to $d(v,u)$ in this case.

Lecture 4

Eulerian circuits, Hamiltonian cycles.

Eulerian circuits

Digraph: A digraph, short for directed graph, is a graph in which the edges have directions. Here we denote the edge from vertex u to v with $\overrightarrow{uv}$.
Outdegree: The outdegree of a vertex, denoted by $d^+(v)$, is the number of edges going out of v, that is, edges of the form $\overrightarrow{vu}$.
Indegree: The indegree of a vertex, denoted by $d^-(v)$, is the number of edges coming into v, that is, edges of the form $\overrightarrow{uv}$.
Trivial component: Trivial components are components of the graph that have no edges. These are also called isolated vertices.
Eulerian circuit: An Eulerian circuit in a graph G is a closed trail (or circuit) in the graph that goes through every edge of G. Since its a trail, we are also saying here that the edge is in the circuit exactly once.
Eulerian trail: An Eulerian trail in a graph G is a trail that goes through every edge of G.
Even graph: A graph is said to be even if the degree of every vertex is even.

Lemma: Every non-trivial even graph contains a cycle.
Proof sketch: Consider a maximal path in the graph. Let u be the first vertex of the path. Since $d(u) > 1$ and this is a maximal path, it must have another neighbor on the path. This will give us the required cycle.

Theorem: a graph G has an Eulerian circuit if and only if G is even and has only one non trivial component.

Proof sketch: If a graph has an Eulerian circuit, then as we walk on this circuit, suppose the edge (u,v) is traversed from u to v. Then direct the edge (u,v) as $\overrightarrow{uv}$. This way, we will get a digraph. The Eulerian circuit tells us that $d^+(v) = d^-(v)$ for every vertex. This would imply that $d(v)$ was even in the original graph.
There are two ways to see the converse, that is, if the graph is even with one non trivial component, then it has an Eulerian circuit. The first proof is using Induction, the second algorithmic.
Inductive proof: We will use induction on the number of edges in the graph. If $m = |E(G)| = 0$, then the graph trivially has an Eulerian circuit. For $m > 0$, then by the lemma above, we can find a cycle C in G. Removing the edges of this cycle may disconnect the graph and there might be more than one component, but each component will in turn be an even graph and with induction hypothesis will contain an Eulerian circuit. Also, since the original graph had one non-trivial components, all these components must share a vertex with the cycle. Then to get the Eulerian circuit for G, we walk along the cycle C but whenever we reach a vertex belonging to some component, we walk along its Eulerian circuit before returning to the cycle.
Sketch of the algorithm: We can also see this proof as an algorithm. Pick any starting vertex v. We start a walk here at v making sure we walk on every edge exactly once. Since the degree of every vertex is even, the only time we cannot walk further will be if we reach back v and we have walked on all edges incident onto v. If all the edges have been walked on, then this was the required Eulerian circuit. If not, we can pick any vertex, all whose edges have not been walked on, and start this process again. In the end, we will join all these walks to get the required Eulerian circuit.

Corollary: A graph G has an Eulerian trail if and only if it has only one non trivial component and exactly two vertices of odd degree.
Proof sketch: if a graph has an Eulerian trail, we again create a digraph as above and the result follows. For the converse, we connect the two odd degree vertices with an edge to get an Eulerian trail. Deleting this extra edge will give us the required Eulerian trail.

Hamiltonian paths, cycles

Spanning cycle: A spanning cycle in a graph G is a cycle that goes through every vertex of the graph. It is also called a Hamiltonian cycle.
Traveller's Dodecahedron game by Sir William Hamilton: finding spanning cycle for any initial path of length 4 in a dodecahedron.
Hamiltonian path: A Hamiltonian path in a graph G is a path that goes through every vertex of the graph.
Hamiltonian graph: Is a graph that contains a Hamiltonian cycle.

Theorem: (Necessary condition for containing a Hamiltonian cycle): If a graph G is Hamiltonian, then $\forall S \subset V(G)$, the graph $G \setminus S$ has atmost $|S|$ components.

Proof sketch: Let C be the Hamiltonian cycle in G. Deleting the vertices in S creates some components in G. The Hamiltonian cycle C cannot go between these components and must go from any component to S before travelling to another component. Let us mark all the arrival vertices into S, that is, every time the cycle goes from a vertex in a component in $G \setminus S$ to a vertex in S, we will mark this vertex. We notice that there must be at least as many distinct arrival vertices as components in $G \setminus S$.

Theorem: (Necessary condition for containing a Hamiltonian **path**): If a graph G contains a Hamiltonian path, then $\forall S \subset V(G)$, the graph $G \setminus S$ has atmost $|S| + 1$ components.

Proof sketch: The Same proof as for the previous statement, only here at most one component need not have an arrival vertex.

Examples of graphs which satisfy the necessary condition but don't contain a Hamiltonian cycle. (Peterson graph is one such).
Minimum degree $\delta(G)$: is the minimum of all degrees in the graph.
Maximum degree $\Delta(G)$: is the maximum of all degrees in the graph.

Theorem (Dirac's): (sufficient condition for containing a Hamiltonian cycle): if G is a graph such that $|V(G)| \geq 3$, and $\delta(G) \geq n/2$, then G is Hamiltonian.

Proof sketch: For a proof by contradiction, assume that G satisfies the conditions mentioned in the theorem, but does not contain a Hamiltonian cycle. Let $(u, v) \notin E(G)$. If after adding the edge (u, v), the graph still does not contain a Hamiltonian cycle, then add it. We repeat this process until adding any further edges will make the graph contain a Hamiltonian cycle. Let this saturated graph be H. We notice that H also satisfies the conditions in the theorem. We will also note that H must contain a Hamiltonian path, otherwise we could have added more edges while not creating a Hamiltonian cycle. Let the vertices on the Hamiltonian path be $u_1, u_2, ...u_n$. Let $I = \{i | (u_1, u_i) \in E(H)\}$ and let $J = \{i + 1 | (u_i, u_n) \in E(H)\}$. Since H is not Hamiltonian, u_1 and u_n are not adjacent. So indices in I and J are between 2 and n, a total of $n - 1$ indices, while $d(u_1) + d(u_2) \geq n$, so $I \cap J \neq \phi$. From this we can construct the Hamiltonian cycle. This contradiction will imply that our original assumption, that G is not Hamiltonian, is false.

Theorem (Ore's): (sufficient condition for containing a Hamiltonian cycle): if G is a graph such that $|V(G)| \geq 3$, and for any two non-adjacent vertices u and v, $d(u) + d(v) \geq n$, then G is Hamiltonian.

Proof is exactly the same as the proof for Dirac's.

A cycle graph (graph that is a cycle) has a Hamiltonian cycle, but does not satisfy degree requirements of Dirac's or Ore's theorem. So this is an example of when the conditions of Dirac's or Ore's theorem are not met, but the graph is still Hamiltonian.

Lecture 5

Bipartite graphs, Chromatic number, Interval graphs.

Bipartite graphs

Lemma: Every odd closed walk (a closed walk with an odd number of edges) contains an odd cycle.

Proof sketch: If no vertex repeats in the walk, then it is the required odd cycle. On the other hand, if a vertex u repeats, and if v is the starting vertex of the closed walk, then u splits the closed walk into two closed walks and one of them is of odd length, and we can repeat this process with this smaller odd closed walk until we get an odd cycle.

Theorem: A graph is bipartite if and only if it does not contain any odd cycles (cycles whose length is odd).

Proof sketch: Given a bipartite graph, any cycle must alternate between the partitions, so every cycle is even. For the converse, given a graph without any odd cycle, we start at a vertex, say s, and run the BFS algorithm. Then we assign every vertex v to partition 1 if its depth is even and to partition 2 if its depth is odd. We know that for any edge (u, v) in the graph $|d(u) - d(v)| \leq 1$ where $d()$ is the depth index. The graph will not be bipartite then if there is an edge (u, v) such that $d(u) = d(v)$. But then walking in the BFS tree from s to u, then on this edge to v and then in the BFS tree back from v to s is an odd closed walk and hence it would imply that the graph has an odd cycle. This is a contradiction, so the graph is bipartite.

Chromatic number

Clique number $\omega(G)$: is the size of the largest clique (complete subgraph) in a graph.
Proper coloring: Given a graph G and a coloring $f : V(G) \to \{1, 2, .., k\}$ with k colors is said to be a proper coloring of G, if neighbors get different colors, that is, $(u, v) \in E(G)$, then $f(u) \neq f(v)$.
Chromatic number $\chi(G)$: is the minimum number of colors required to obtain a proper coloring of G.
Greedy algorithm for coloring: Take any ordering of the vertices as $v_1, v_2, ...v_n$. We color the vertices in the order of their index using the first available color from a list of colors.

Theorem: For any graph G, $\chi(G) \leq \Delta(G) + 1$.

Proof sketch: In the greedy algorithm, a vertex is connected to at most $\Delta(G)$ vertices before it, so $\Delta(G) + 1$ colors is sufficient to give it a proper coloring.

Theorem: For any graph G, $\chi(G) \geq \omega(G)$.
Proof: need a different color for every vertex of a maximum clique.

Zykov's construction: We iteratively construct graphs $G_3, G_4, ...$ such that $\forall i$, $\omega(G_i) = 2$, while $\chi(G_i) = i$. First, we let G_3 be a 5-cycle. Then assuming that G_k has been constructed, we construct G_{k+1} by taking k disjoint copies of G_k, call them $H_1, H_2, ..., H_k$ and we also add a set S of vertices such that for every set $\{v_1, v_2, ..., v_k\}$ of vertices where $v_i \in H_i$, there is a distinct vertex in S whose only neighbors are $\{v_1, v_2, ..., v_k\}$. We can see that k colors are not sufficient to color G_{k+1} because every one of the k colors will appear in each copy H_i and so it is not possible to color S. Also, we can check there will be no triangles.

Interval graphs

Interval graphs: Consider a graph whose vertices are closed finite intervals on the real number line and any two vertices are adjacent if and only if their corresponding intervals intersect.

Theorem: For any interval graph G, $\chi(G) = \omega(G)$.
Proof sketch: order the vertices of G according to increasing leftmost endpoints of their corresponding intervals. We will use the greedy algorithm with this order of vertices. We will note that if any vertex v_i has k neighbors with index less than i, then the intervals corresponding to all these neighbors contain the leftmost endpoint of the interval corresponding to v, so they all form a clique of $k + 1$ vertices. So if the greedy algorithm takes m colors to color, then $\omega(G) = m$.

Lecture 6

Planar graphs, Euler's Formula, Kuratowski's Theorem, Dual.

Planarity

Drawing: A drawing of a graph in the plane $\mathbb{R}^2$ is where vertices are represented by points, and edges by continuous curves that don't go through any other vertex besides its endpoints.
Planar embedding: A planar embedding of a graph is a drawing of in which none of the edges intersect at any other point besides their endpoints.
Planar graph: A graph G is said to be planar if it has a planar embedding.
Plane graph: A fixed planar embedding of a planar graph is called a plane graph.
Face (connected regions): Given any plane graph, its vertices and edges divide the plane $\mathbb{R}^2$ into connected regions. These are the faces of the plane graph. When a plane graph is connected, then the boundary of each face is a closed walk (a walk that starts and ends in the same vertex). When it is disconnected, then there are faces whose boundary consists of more than one closed walk.
Length of a face: is the total length of closed walk(s) bounding the face, denoted by $l(F)$ where F is a face of the plane graph.
Dual of a plane graph: The dual of a plane graph G is G^*, where its vertices are faces and for every edge in G we draw an edge in G^* connecting the faces it bounderies. Note that edges which have the same face on both sides create a loop at the vertex of that face.
Observation: Dual graphs are planar and connected. $n^* = f$, $e^* = e$ and $f^* = n$.

Theorem: For any plane graph G, $\sum l(F_i) = 2e$.
Proof sketch: if G^* is the dual of G, then $l(F_i)$ is the degree of vertex corresponding to F_i in G^*, and we note that $e = e^*$.

Euler's Formula

Theorem: (Euler's Formula): For any connected plane graph G, let $n = |V(G)|$, $e = |E(G)|$ and let f denote the number of faces. Then, $n - e + f = 2$.
Proof sketch: We will use induction on n. If $n = 1$, then graph has only loops $e = f - 1$. For $n > 1$ contract an edge (u, v) to form a single vertex. Then we notice that $n' = n - 1$, $e' = e - 1$, and $f' = f$. The result follows using Induction hypothesis for G'.

Theorem: (Euler's Formula for disconnected graphs): For any plane graph G with k components, let $n = |V(G)|$, $e = |E(G)|$ and let f denote the number of faces. Then, $n - e + f = k + 1$.
Proof sketch: Let the components of G be $G_1, .., G_k$, then in a planar embedding of G, all these components will only share the outer face. Also, $n_i - e_i + f_i = 2$, summing all these, $\sum n_i - \sum e_i + \sum f_i = 2k$, and this gives $n - e + (f + k - 1) = 2k$. $\sum f_i = f + k - 1$ because the outer face is counted once in each component, but should be counted just once for G. Simplifying this gives the required equation.

Observation: any graph that can be drawn on the sphere without crossings can be drawn on the plane also without crossings. In particular, the graphs of the regular polyhedra are planar.

Theorem: There are only 5 regular polyhedra.
Proof sketch: since the polyhedra are planar, we use Euler's formula and note that if the polyhedra is k-regular (degree of every vertex is k), and if every face is bounded by l edges, then $\frac{2e}{k} - e + \frac{2e}{l} = 2$. But $k, l \geq 3$. So in $\frac{2}{k} + \frac{2}{l} = \frac{2}{e} + 1$, or $\frac{1}{k} + \frac{1}{l} > \frac{1}{2}$. we can check values of k, l to find the five regular polyhedra.

Theorem: For any simple planar graph G, $e \leq 3n - 6$. Further if G does not have any triangles, then $e \leq 2n - 4$.
Proof sketch: fix a planar embedding of G. Here, sum of lengths of faces is equal to $2e$. Since G is simple, sum of length of faces is at least $3f$, so $3f \leq 2e$. We plug this in Euler's formula. No triangles will imply $4f \leq 2e$.

Theorem: K_5 and $K_{3,3}$ are not planar.
Proof sketch: They have 10 and 9 edges respectively, but if they were planar, then using the previous theorem, they can have at most 9 and 8 edges respectively thereby giving a contradiction.

Homomorphism, Kuratowski's Theorem

Subdivision: A subdivision of a graph G is a graph G' obtained by replacing edges of G with disjoint paths. We observed that subdivisions of K_5 and $K_{3,3}$ are also not planar, since, if they were planar we could replace the paths with edges to get a drawing of K_5 and $K_{3,3}$.
Homeomorphic graphs: A graph G is homeomorphic to G' if some subdivision of G is isomorphic to some subdivision of G'. (Think of this as the two graphs having the same underlying base graph).

Theorem: (Kuratowski's Theorem): A graph G is planar if and only if it does not contain a subgraph homeomorphic to K_5 or $K_{3,3}$.

Strategy for problems: To show that a graph is planar, give a drawing. To show that it is not planar, either show that it has more than $3n - 6$ edges (or $2n - 4$ if triangle free), or find a subgraph of the graph that looks like a subdivision of K_5 or $K_{3,3}$.

Theorem: (Fary's Theorem): Every simple planar graph has a planar embedding with only straight line edges.

Lecture 7

Vertex/Edge covers τ, ρ, Independent sets α, ν, Gallai's Theorems, Kruskal's algorithm.

Homomorphism, Kuratowski's Theorem

All results/definitions of this lecture are for simple graphs.

$\tau(G), \nu(G)$

Independent set of edges or a Matching: A subset of the edges of a graph is called a matching or an independent set if no two edges share a common endpoint.
Maximal matching: A matching is said to be maximal if we cannot add any more edges to it to obtain a larger matching. **Maximum** matching: A matching is said to be maximum if it is a matching in the graph with the largest number of edges, that is, there is no other matching in the graph with strictly more number of edges.
$\nu(G)$: $\nu(G)$ is the size (number of edges) of a maximum matching in G.
Vertex cover: A vertex cover is a set $S \subset V(G)$ of **vertices** covering all the **edges** of G. An edge is said to be covered if at least one of its endpoints is in S. (Minimal and minimum defined as for maximal and maximum).
$\tau(G)$: $\tau(G)$ is the size (number of vertices) of a minimum vertex cover of G.

Theorem: For any simple graph G, $\nu(G) \leq \tau(G)$.
Proof sketch: Given any matching in G, a vertex cover will require to contain at least one endpoint of each edge in the matching.

$\alpha(G), \rho(G)$

Independent set of vertices: A subset of the vertices of the graph is said to be an independent set of vertices if no two vertices in this subset are adjacent (have an edge between them).
$\alpha(G)$: $\alpha(G)$ is the size (number of vertices) of a maximum independent set in G.
Edge cover: A set $S \subset E(G)$ of **edges** covering all the **vertices** of G is said to be an edge cover of the graph. A vertex is said to be covered if there is an edge from S incident onto it.
Note: A graph with isolated vertices does not have an edge cover. So whenever we talk about edge covers, we assume we are talking about a graph without isolated vertices.
$\rho(G)$: $\rho(G)$ is the size (number of edges) of a minimum edge cover of G.

Theorem: For any simple graph G, $\alpha(G) \leq \rho(G)$.
Proof sketch: We notice that given any independent set of vertices, each vertex in it will require a distinct edge to cover it.

Gallai's theorems

Theorem: For any set $S \subset V(G)$, S is an independent set of vertices if and only if $V(G) \setminus S$ is a vertex cover.
Proof sketch: if S is an independent set of vertices, every edge is adjacent to atleast one vertex in $V(G) \setminus S$. On the other hand, if S is a vertex cover, $V(G) \setminus S$ cannot contain any edges.

Theorem: Given a simple graph G and a set of S of k edges. Then,
(a) If S is a matching, then there is an edge cover in G with at most $n - k$ edges.
(b) If S is an edge cover, then there is a matching in G with at least $n - k$ edges.

Proof sketch: (a) S covers $2k$ vertices. We pick one edge for every vertex that is not covered by S, this takes at most $n - 2k$ more edges.
(b) Let $H = (V(G), S)$ be the subgraph of G consisting of only the edges of S. Let H have c components. Then H has at least $n - c$ edges (why?). So $n - c \leq k$ or $c \geq n - k$. Chose one edge from each component for the matching to get a matching of size $n - k$.

Theorem: (Gallai's Theorem): For any simple graph G,
(a) $\alpha(G) + \tau(G) = n$
(b) if G doesn't have any isolated vertices, then $\nu(G) + \rho(G) = n$.
Proof sketch: (a) If S is a maximum independent set of size $\alpha(G)$, then $V(G) \setminus S$ is a vertex cover, so $\tau(G) \leq n - \alpha(G)$. Similarly if S is a minimum vertex cover, then $V(G) \setminus S$ is an independent set of vertices, so $\alpha(G) \geq n - \tau(G)$. We get the required result by combining both these inequalities.
(b) If S is a maximum independent set of edges size $\nu(G)$, then there is an edge cover of size at most $n - \nu(G)$, so $\rho(G) \leq n - \nu(G)$. Similarly if S is a minimum edge cover of size $\rho(G)$, then there is a independent set of edges of size at least $n - \rho(G)$, so $\nu(G) \geq n - \rho(G)$. We get the required result by combining both these inequalities.

Kruskal's algorithm

Minimum weight spanning tree: Given a graph G and an edge weight function $w : E(G) \to \mathbb{R}^+ \cup 0$, a minimum weight spanning tree is defined as a spanning tree with the least total weight (sum of weights of the edges of the spanning tree).

Kruskal's Algorithm for finding the minimum weight spanning tree: Greedy algorithm of always adding the smallest weight edge that does not form a cycle (so it will connect two components).

Proof sketch: Let $e_1, e_2, ..., e_m$ be an ordering of the edges with increasing weight, that is $w(e_i) \leq w(e_{i+1})$. Let $M_i = 0$ if e_i is not in the spanning tree found by Kruskal's algorithm, and $M_i = 1$ if it is in it. We define O_i similarly for a minimum spanning tree of the graph. Further if k is the largest i such that $M_i = O_i$ for $1 \leq i \leq k$, then we pick a minimum spanning tree such that this k is the largest possible. We now show that $k = m$, as this will show that the spanning tree returned by the kruskal's algorithm is also a minimum weight spanning tree. Let $k \neq m$, and $M_{k+1} \neq O_{k+1}$. If e_{k+1} is in the minimum weight spanning tree, then it does not induce a cycle with the previously picked edges, so it is not possible that $M_{i+1} = 0$. On the other hand, if $M_{i+1} = 1$ and $O_{i+1} = 0$, then we can add the edge e_{i+1} to the minimum weight spanning tree. It will create a cycle with only edges $e_j, j > i + 1$, so removing another edge from the cycle will give us a minimum weight spanning tree with a longer matching sequence, or a larger k, which is a contradiction.

Input: A connected graph $G = (V, E)$ with n vertices and m edges and a weight function $w : E \to \mathbb{R}$.

1. ORDER the edges in the order of increasing weight: $w(e_1) \leq w(e_2) \leq ... \leq w(e_m)$

```
2. $k \leftarrow 0$; $j \leftarrow 1$; $E(F) \leftarrow \phi$
3. **loop: while** $k < n - 1$
4.     Check if the endpoints of $e_j$ are in different components.
5.     **if** they are in different components, **then:**
6.         $E(F) \leftarrow E(F) \cup e_j$
7.         $k \leftarrow k + 1$
8.     $j \leftarrow j + 1$
9. **end loop**
```

Lecture 8

Matchings in Bipartite graphs, Augmenting path algorithm, Hall's and Frobenius's Theorem

Augmenting path algorithm

We will denote bipartite graphs with $G(A, B, E)$, where A is the first partition, B the second, and E the set of edges going between them.
M-Augmenting path: Given a bipartite graph $G(A, B, E)$ and a matching M in it, an M-augmenting path satisfies the following three conditions:

1. The path starts in an unmatched vertex
2. The path ends in an unmatched vertex
3. The edges of the path alterate between edges not in the matching and edges from the matching.

Notice that if the M-augmenting path starts in the partition A, then it must end in B. If we make this a directed path starting from the vertex in A, then every edge of the path going from A to B will be an edge not in the matching, while every edge from a vertex in B to A will be an edge of the matching.
M-alternating path: The definition of M-alternating path is same as above, but it is only required to satisfy condition 1 and 3.
Hungarian mathematician Kőnig Denes' augmenting path algorithm:

Input: Bipartite graph $G(A, B, E)$ and a matching M. Output: either a M-augmenting path, and if it doesn't exist, then a minimal vertex cover.

```
1. Initialization: Let $U$ be the set of unmatched vertices in $A$. Set $S = U$ and $T = \phi$. Initially all
   vertices in $S$ are unmarked and $p(v) = *, \forall v$.
2. **loop: while** there is an unmarked vertex $x \in S$,
3.     $\forall xy \in E$ such that $xy \notin M$,
4.         if $p(y) = *$, set $p(y) = x$
5.         if $y$ is unmatched, return $y, p(y), p(p(y))...$ as an $M$-augmenting path.
```

```
6.            if $y$ is matched so that $zy \in M$, then,
7.                add $y$ to $T$
8.                add $z$ to $S$
9.                if $p(z) = *$ then set $p(z) = y$
10.       mark $x$
11. **end loop**
12. return $T \cup (A \setminus S)$ as a minimum vertex cover.
```

We can now use this as a subroutine to find the maximum matching.

```
Input: Bipartite graph $G(A, B, E)$.
1. Initialization: Set $M = \phi, S = \phi, T = \phi$.
2. Run the $M$-augmenting path subroutine.
3. **loop: while** there is an augmenting path,
4.     swap the edges of the augmenting path in the matching $M$ with the ones not in $M$
5.     run the $M$-augmenting path algorithm with the new matching.
6. Return $M$ as the maximum matching and $T \cup (A \setminus S)$ as the minimum vertex cover.
```

Theorem: The set $T \cup (A \setminus S)$ given by the algorithm is the minimum vertex cover.
Proof sketch: there are no edges between S and $B \setminus T$, so it is a vertex cover. All vertices in $T \cup (A \setminus S)$ are matched, but there is no matching edge between T and $A \setminus S$. So, size of this vertex cover is equal to the size of the matching M.

Halls theorem, Frobenius theorem

Hall's and Frobenius theorems give sufficient conditions for matchings in bipartite graphs.

Hall's Theorem: Given a bipartite graph $G(A, B, E)$, there is a matching of A into B (that is, all vertices of A are matched into B), if and only if $\forall S \subset A$, $|S| \leq |N(S)|$, where $N(S)$ is the neighborhood of S in B.
Proof sketch: If G has a matching, then the claim is obvious. For the other direction, we prove the countrapositive of the sentence. So we show that if G does not have a matching of A into B, then there must be a set S with $|S| > |N(S)|$. For this, we run the augmenting path algorithm with some maximum matching. Since A cannot be matched into B, there are some unmatched vertices in A, but there is no augmenting path. We look at the sets S and T returned by the algorithm. We note that $|S| > |T|$, since S has some unmatched vertices, while all vertices of T are matched. But $N(S) = T$.

Frobenius' Theorem: A bipartite graph $G(A, B, E)$ contains a perfect matching if and only if

1. $|A| = |B|$, and,
2. $\forall S \subset A$, $|S| \leq |N(S)|$, where $N(S)$ is the neighborhood of S in B.

Proof sketch: Corollary of Hall's.

Theorem: Every regular bipartite graph $G(A, B, E)$ has a perfect matching.
Proof sketch: We will show that all conditions of Frobenius' theorem are met. Let the degree of every vertex be k. By counting the number of edges in the graph by counting the degrees of vertices in A, $|E| = |A|k$. But also, $|E| = |B|k$, so $|A| = |B|$.
Pick a set $S \subset A$ of vertices. Let $N(S)$ be its neighborhood. Consider the graph restricted to only the vertices S and $N(S)$ and the edges between them. All vertices in S here will have degree k, but vertices in $N(S)$ have degree at most k. So $k|S| =$ number of edges $\le k|N(S)|$.

Lecture 9

Edge chromatic number, Vizing's theorem, Konig's theorem, Maximum flows in networks.

Edge chromatic number

We will consider loopless graphs for the edge coloring problem
Proper edge coloring: A coloring of the edges of a graph is said to be proper if adjacent edges (edges with a common end point) get different colors.
Observation: In any proper edge coloring of a graph, the edges in one color class form a matching or an independent edge set.
$\chi_e(G)$: $\chi_e(G)$ is defined as the minimum number of colors required in a proper edge coloring of the graph.

(Trivial) Theorem: For any loopless graph, $\Delta(G) \le \chi_e(G) \le 2\Delta(G) - 1$.
Proof sketch: the $\Delta(G)$ edges of a vertex with degree $\Delta(G)$ are all adjacent to each other and require different colors. On the other hand, any edge is adjacent to at most $2\Delta(G) - 2$ edges, so $2\Delta(G) - 1$ colors are sufficient.

Theorem: (Vizing's): For any **simple** graph, $\chi_e(G) \le \Delta(G) + 1$.

Examples: We saw that $\chi_e(K_{2n-1}) = \chi_e(K_{2n}) = 2n - 1$.

Theorem: (Shannon's Theorem): For any loopless graph, $\chi_e(G) \le \frac{3}{2}\Delta(G)$.

Examples: Fat triangle: Triangle with each edge replaced by 3 parallel edges. We see that this requires 9 colors and $\Delta(G) = 6$.

Theorem: (Kőnig's): For a bipartite graph $G(A, B, E)$ (loopless but not necessarily simple) $\chi_e(G) = \Delta(G)$.
Proof sketch: By induction on $\Delta(G)$. Base case: If $\Delta(G) = 1$ then the graph is a matching and one color suffices to color all edges. Now assuming $\Delta(G) > 1$, if G is regular, then it must contain a perfect matching. Deleting the edges of this matching will give a graph with maximum degree $\Delta(G) - 1$ and by induction hypothesis, it requires $\Delta(G) - 1$ colors. We give the perfect matching one additional color to color edges of G with $\Delta(G)$ colors.
If G is not regular, we will first add isolated vertices to the partition with fewer vertices and make both partitions contain equal number of vertices. Then, if there is a pair of vertices $u \in A$ and $v \in B$, such

that $d(u) \neq \Delta(G)$ and $d(v) \neq \Delta(G)$, then we connect them by an edge. We repeat this process until the graph is $\Delta(G)$-regular. Double counting the edges by counting the degrees of A and B will reveal that if there is a vertex $u \in A$ such that $d(u) \neq \Delta(G)$, then there must be such a v in the partition B also.

Maximum flows in networks.

Network: A network is a digraph (directed graph) $G(V, E)$, with a non-negative capacity function $c : E(G) \to \mathbb{R}^+ \cup \{0\}$, and two distinguished vertices, s called source, and t, called the sink. It is denoted by the tuple (G, s, t, c).
We refer to the vertices of the graph that are not source or sink as nodes.
Flow: A flow on a network is a function $f : E(G) \to \mathbb{R}^+ \cup \{0\}$, such that,

- (Capacity constraint) $0 \leq f(e) \leq c(e)$, and,
- (Conservation of flow) $f^+(v) = f^-(v), \forall v \in V(G)$,

where we define $f^+(v) = \sum_{e=\overrightarrow{vu}} f(e)$ is the out-flow from a vertex, and $f^-(v) = \sum_{e=\overrightarrow{uv}} f(e)$ is the in-flow into a vertex.

Value of a flow: The value of a flow is $val(f) = f^-(t) - f^+(t) = f^+(s) - f^-(s)$, or the new flow out of the source and into the sink.
We study the problem of finding the maximum flow in a network. The following is the Edmond-Karp's algorithm for finding the maximum flow in a network. We do this by keeping track of possible increments/decrements in flow using the auxilary graph. We notice that if $f(e)$ is the flow through an edge e, then we can increase the flow on e by $c(e) - f(e)$, and we can reduce the flow on edge e by $f(e)$.

Auxillary graph: Given a network (G, s, t, c) and a flow f on it, the auxillary graph H_f is a directed graph on $V(G)$. For each edge $e = \overrightarrow{uv}$,

- If $f(e) < c(e)$, then the auxillary graph will contain edge $\overrightarrow{uv}$ with label $c(e) - f(e)$ denoting the amount flow can be increased. We will call this a **forward edge**.
- If $f(e) > 0$, then it will contain the edge $\overrightarrow{vu}$ with label $f(e)$ to denote the amount by which flow on edge e can be decreased. We will call this is a **backward edge**.

Flow Augmenting Path: Given a network (G, s, t, c), a flow f, and the corresponding auxillary graph H_f, a flow augmenting path is a directed path in H_f from s to t.
We note that if there is a flow augmenting path, then the flow from s to t can be increased by the minimum value on the edges of this path.
We note further that the augmenting path algorithm was originally discovered by Ford-Fulkerson. The modification of the algorithm to consider the shortest augmenting flow in the BFS on the auxillary graph is called the Edmond Karp's algorithm.

Edmond Karp's algorithm

Pseudocode for Edmond Karps algorithm:

Edmond Karp's algorithm for maximum flow in a network. Input: Network (G, s, t, c).

```
1. For every $e \in E(G)$, set $f(e) = 0$. Set $S = \phi$. (Note: initialization can be with a non-zero flow).
2. Loop:
3.     Construct the Auxillary graph $H_f$
4.     Run the BFS algorithm on $H_f$ to find a flow augmenting path.
5.     If there is a flow augmenting path,
6.         Let $\delta$ be the minimum label on the edges of the flow augmenting path.
```

```
7.            For each forward edge on the flow augmenting path, for its corresponding edge $e$ in $G$,
    set $f(e) \leftarrow f(e) + \delta$.
8.            For each backward edge on the flow augmenting path, for its corresponding edge $e$ in $G$,
    set $f(e) \leftarrow f(e) - \delta$.
9.        If there is no flow augmenting path,
10.           Update $S$ to be the subset of vertices reachable from $s$ in $H_f$ and end loop.
11. return $f, S$.
```

Lecture 10

Minimum cut, Max Flow-Min Cut Theorem, Vertex/Edge-Connectivity.

s,t-cut, capacity of cuts

s,t-cut: Let $S \subset V(G)$, such that $s \in S$ and $t \notin S$. Then S is called a s,t-cut.
Capacity of a cut: Given S a s,t-cut, its capacity is gien by $\sum_e c(e)$ where the sum is over all edges e such that $e = \overrightarrow{uv}$ with $u \in S$ and $v \notin S$.
Flow across a cut S: Given a network and a flow f on it, we define the flow across the cut S as

$$\sum_{e=\overrightarrow{uv}, u\in S, v\notin S} f(e) - \sum_{e=\overrightarrow{uv}, u\notin S, v\in S} f(e)$$

the sum of all flow leaving S minus the sum of all flow coming into S.

Observation: Given a flow f and S a s,t-cut, the flow across S equals $val(f)$, the value of the flow out of s.
Proof sketch: Consider $\sum_{u\in S} f^+(u) - f^-(u)$. Because of conservation of flow, this sum is just $val(f)$ or $f^+(s) - f^-(s)$. But also, for every edge $e = \overrightarrow{uv}$ such that both $u, v \in S$, this edge would count in $f^+(u)$ and also in $f^-(v)$ and will therefore cancel off. So the only terms that will remain will be for edges $e = \overrightarrow{uv}$ such that one endpoint is in S and the other is not. This gives us precisely the flow across the cut.

Observation: Value of maximum flow $\leq$ Capacity of minimum cut.
Proof sketch: The flow across any cut is maximum if the edges out of the cut carry as much flow as $c(e)$ while the edges bring flow in carry 0 flow. But this is the capacity of the cut.

Ford-Fulkerson's theorem

Theorem: (Ford-Fulkerson, 1956): In every network, the maximum value of a feasible flow equals the minimum capacity of a source/sink cut.
Proof sketch: We run the Edmond-Karp's algorithm until there is no augmenting flow path. Then the algorithm returns a set S of all vertices that can be reached from s in the auxillary graph. Since there is not augmenting flow path, $t \notin S$. Further if $e = \overrightarrow{uv}$ such that $u \in S$ and $v \notin S$, then $f(e) = c(e)$,

otherwise the auxillary graph would contain a forward edge and v will need to be in S. Similarly, if $e = \overrightarrow{uv}$ such that $u \notin S$ and $v \in S$, then $f(e) = 0$, otherwise the auxillary graph would contain a backward edge from v to u and u will need to be in S. So then the flow across S is its capacity. But then $val(f) =$ capacity of the cut S. Then this has to be the maximum flow and minimum cut respectively and their values agree.

Lemma: (Integrality Lemma): if $\forall e \in E(G), c(e) \in \mathbb{Z}^+$, then, there exists a maximum flow taking integer values on all edges, i.e., $\forall e \in E(G), f(e) \in \mathbb{Z}^+$.
Proof sketch: the increment δ in the Edmond-Karp's algorithm, if run starting with the 0 flow, will always be an integer.

MIDTERM SYLLABUS ENDS HERE.

Modifications of the network flow problem

We will consider three modifications of the network flow problem.

Modification 1: multiple sources $s_1, s_2, ..., s_k$ and multiple sinks $t_1, t_2, .., t_l$.
To solve this problem we add two new nodes, a super source S and a super sink T. We add edges with ∞ capacity from S to each s_i and from each t_j to T. Any solution to this will give us a solution to the original problem.

Modification 2: Some vertices can have a capacity constraint of the maximum flow that can pass through it, denoted by $c(v)$.
For every v with a capacity $c(v)$, we add two new vertices v_{in} and v_{out}. v_{in} has all the incoming edges of v, while v_{out} has all the outgoing edges of v. We also add one edge directed from v_{in} to v_{out} with capacity $c(v)$. We can easily see that a solution to this problem will give us a solution to the original problem.

Modification 3: we are given an undirected graph with edge-capacities.
We can solve this problem by constructing a new directed graph G' with the same vertex set as G. For each edge $e = uv \in E(G)$ with capacity $c(e)$, G' will have two directed edges, $\overrightarrow{uv}$ and $\overrightarrow{vu}$, both with capacity $c(e)$. Again, its easy to see that a solution to this problem will provide a solution to the original.

Lecture 11

Vertex/Edge-Connectivity, Menger's Theorems.

Vertex and edge connectivity

k-connected: A graph G is said to be k-connected, or k vertex-connected, if deleting any $k-1$ vertices from the graph does not disconnect it.
Vertex-connectivity: $\kappa(G)$ or the vertex connectivity number of the graph G (where G is not the complete graph) is defined as the largest k for which G is k-vertex-connected. Since K_n is connected after removal of any number of vertices, we adopt the convention that $\kappa(K_n) = n - 1$.

Observation: $\kappa(G) \leq \delta(G)$.
Proof sketch: deleting all the vertices adjacent to the vertex with minimum degree will either disconnect the graph, or give one isolated vertex. In the later case, then we know that the graph was a complete graph and $\delta(G) = n - 1 = \kappa(K_n)$. In the former case, since we could disconnect the graph with deleting $\delta(G)$ vertices, so $\kappa(G) \leq \delta(G)$.

k-edge-connected: A graph G is said to be k-edge-connected, if deleting any $k-1$ edges from the graph does not disconnect it.
Edge-connectivity: $\lambda(G)$ or the edge connectivity number of the graph G is defined as the largest k for which G is k-edge-connected.

Theorem: $\kappa(G) \leq \lambda(G) \leq \delta(G)$.
Proof sketch: The first inequality is true because we can always remove one end point of each edge in a disconnecting edge set to disconnect the graph. This way, we are removing at most $\lambda(G)$ edges.
On the other hand, removing all the edges incident on a vertex v with $d(v) = \delta(v)$ disconnects the graph, giving the second inequality.

Edge-cuts and Edge-disjoint paths

Lemma: Let (G, s, t, c) be a directed graph with capacity 1 assigned to each edge $c(e) = 1 \forall e \in E(G)$. If there is a flow f such that $val(f) = d > 0$ and f takes a value of 0 or 1 on every edge, then G contains d edge-disjoint paths from s to t.
Proof sketch: We use induction on d. For $d = 1$, there is one path from s to t. Because of conservation of flow at every vertex, a path starting at s and walking on only edges with flow 1 must end at t. For $d > 1$, consider G_f to be the graph of edges with $f(e) = 1$. Since there is a non-zero flow from s to t, this graph has a s,t-directed path. Deleting the edges of this path will give us a subgraph of G_f with a flow of $d-1$ from s to t.

Note: G_f as defined above may have more edges than the paths we found.
Disconnecting set of edges or an edge cut: Given $x, y \in V(G)$, a subset $S \subset E(G)$ is called a (x, y)-disconnecting set of edges or an edge cut separating x from y, if deleting the edges in S will disconnect x from y, i.e. there is no path from x to y.
Edge-disjoint paths: Given $x, y \in V(G)$, two x, y-paths are said to be edge-disjoint if they don't share any edges.

Theorem: Given a **directed** graph G with two vertices $s, t \in V(G)$ and let $k \in \mathbb{N}$. Then the following statements are equivalent:

1. There are k edge-disjoint s,t-paths.
2. There is no (s,t)-disconnecting set of edges of size $k-1$
3. (G, s, t, c) has a flow of value at least k, where $c(e) = 1 \forall e \in E(G)$.

Proof sketch: To see that $1 \Rightarrow 2$, if there are k edge-disjoint paths, then a set of $k-1$ edges can only disconnect at most $k-1$ of these paths and there would still be a path from s to t.

To see that $2 \Rightarrow 3$, we notice that if we look at the outgoing edges out of any s,t-cut, then these edges also form a edge-cut or a disconnecting set of edges between s and t. Since they cannot be $k-1$ or less in number, the capacity of any minimum cut is at least k. So by the Ford-Fulkerson theorem, there must be a flow in G of value at least k.

To see that $3 \Rightarrow 1$, we notice that this is the Theorem we proved before.

The above theorem is also true for undirected graphs.

Theorem: Given a **undirected** graph G with two vertices $s, t \in V(G)$ and let $k \in \mathbb{N}$. Then the following statements are equivalent:

1. There are k edge-disjoint s, t-paths.
2. There is no (x, y)-disconnecting set of edges of size $k - 1$
3. (G, s, t, c) has a flow of value at least k, where $c(e) = 1 \forall e \in E(G)$.

Proof sketch: The proof is essentially the same, only we use the Network Flow Modification discussed in the earlier lecture for undirected graphs and will replacing every edge with two directed edges going in both directions and with capacity 1.

Local-edge-connectivity: Given a graph G and two vertices $x, y \in V(G)$, we define $\lambda(x, y)$ as the minimum size of a (x, y)-disconnecting set of edges and $\lambda'(x, y)$ as the maximum number of edge-disjoint x, y-paths.

Menger's theorem for local edge connectivity: Given a graph G, $\forall s, t \in V(G), s \neq t$, $\lambda(s, t) = \lambda'(s, t)$.
Proof sketch: This follows from the previous theorem. For any $s \neq t$, there are $\lambda'(s, t)$ edge-disjoint paths between s and t. Then the size of any edge disconnecting set is at least $\lambda'(s, t)$, so $\lambda(s, t) \geq \lambda'(s, t)$. On the other hand if we don't have a disconnecting set of edges of size $\lambda(s, t) - 1$, then by the previous theorem, we have a flow of value $\lambda(s, t)$ and so there are at least $\lambda(s, t)$ number of edge-disjoint s, t-paths, so $\lambda(s, t) \leq \lambda'(s, t)$.

Vertex-cuts and Vertex-disjoint paths

Disconnecting set of vertices or a vertex cut: Given $x, y \in V(G)$, a subset $S \subset V(G)$ is called a (x, y)-disconnecting set of vertices or a vertex cut separating x from y, if deleting the vertices in S will disconnect x from y, i.e. there is no path from x to y.
Vertex-disjoint paths: Given $x, y \in V(G)$, two x, y-paths are said to be vertex-disjoint if they don't share any vertices.

Theorem: Given a **directed** graph G with two vertices $s, t \in V(G)$ and let $k \in \mathbb{N}$. Then the following statements are equivalent:

1. There are k vertex-disjoint s, t-paths.
2. There is no (x, y)-disconnecting set of vertices of size $k - 1$
3. (G, s, t, c) has a flow of value at least k, where $c(e) = 1 \forall e \in E(G)$ and $c(v) = 1 \forall v \neq s, t$.

Proof sketch: To see that $1 \Rightarrow 2$, if there are k vertex-disjoint paths, then a set of $k - 1$ vertices can only disconnect at most $k - 1$ of these paths and there would still be a path from s to t.

To see that $2 \Rightarrow 3$, we will use the modification of networks discussed earlier to create two vertices v_{in} and v_{out} for each vertex v and draw a directed edge of capacity 1 between them. We notice that since all incoming edges come to v_{in} and all out going edges out of v_{out}, the capacity of a cut is minimized if atleast one neighbor of v_{out} is outside the cut, then we put v_{out} also out of the cut. Similarly if v_{in} was not in the cut, but had atleast one incoming edge from the cut, then we can make the capacity of the cut smaller by putting v_{in} in the partition with s. So then we will notice that we reduce the cut to only have edges going between v_{in} and v_{out}, for different v, and this is precisely a vertex cut. So if there is no vertex cut of size $k - 1$, then the capacity of any minimum cut is at least k. So by the Ford-Fulkerson theorem, there must be a flow in G of value at least k.

To see that $3 \Rightarrow 1$, we notice that this is the Theorem we proved before.

The above theorem is also true for undirected graphs.

Theorem: Given a **undirected** graph G with two vertices $s, t \in V(G)$ and let $k \in \mathbb{N}$. Then the following statements are equivalent:

1. There are k vertex-disjoint s, t-paths.
2. There is no (x, y)-disconnecting set of vertices of size $k - 1$
3. (G, s, t, c) has a flow of value at least k, where $c(e) = 1 \forall e \in E(G)$ and $c(v) = 1, \forall v \neq s, t$.

Proof sketch: it is essentially the same proof as above, only we use the network modification that for each undirected edge, we will create two directed edges, and then for accounting for the vertex capacity, we will split every vertex v into v_{in} and v_{out}.

Local-vertex-connectivity: Given a graph G and two vertices $x, y \in V(G)$, we define $\kappa(x, y)$ as the minimum size of a (x, y)-disconnecting set of vertices and $\kappa'(x, y)$ as the maximum number of vertex-disjoint x, y-paths.

Menger's theorem for local vertex connectivity: Given a graph G, $\forall s, t \in V(G), s \neq t$, $\kappa(s, t) = \kappa'(s, t)$. Proof sketch: This follows from the previous theorem. For any $s \neq t$, there are $\kappa'(s, t)$ vertex-disjoint paths between s and t. Then the size of any vertex-cut is at least $\kappa'(s, t)$, so $\kappa(s, t) \geq \kappa'(s, t)$. On the other hand if we don't have a disconnecting set of vertices of size $\kappa(s, t) - 1$, then by the previous theorem, we have a flow of value $\kappa(s, t)$ and so there are at least $\kappa(s, t)$ number of vertex-disjoint s, t-paths, so $\kappa(s, t) \leq \kappa'(s, t)$.

Menger's Theorems

All the previous theorems can be combined to obtain the following statements about vertex and edge-connectivity numbers of a graph:

Menger's theorem for connectivity: For any (directed) graph and any $k \geq 1$,

1. G is k-edge connected if and only if there are k (directed) edge-disjoint paths between any two distinct vertices.
2. G is k-connected (or k-vertex connected) if and only if there are k (directed) vertex-disjoint paths between any two distinct vertices.

Proof sketch: It follows quite directly from the theorems about local connectivity numbers. If a graph is k-edge connected, then $k - 1$ edges cannot disconnect the graph, then for any two vertices $s \neq t$, $\lambda(s, t) \geq k$, so $\lambda'(s, t) \geq k$. And similarly for vertex disjoint paths.

Lecture 12

Dijkstra's algorithm, Bellman-Ford algorithm.

Dijkstra's Algorithm

Given a weighted graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}^+ \cup \{0\}$ and a vertex $s \in V$, we want to find minimum weight s, v-paths for all vertices $v \in V, v \neq s$. (Here weight of a path is the sum of the weights of the edges in it).
Definition: Let $d(u, v)$ denote the shortest path between the vertices u and v.
We will solve this using Dijkstra's algorithm given below. We will keep track of $t(v)$ the temporary distances of vertices v from s, $p(v)$ will denote the parent of v on the (temporary) shortest path. K is a set of processed vertices. Once a vertex is added to K, $t(v) = d(s, v)$, that is, $t(v)$ will be equal to the length of the shortest s, v-path. So the algorithm ends when $K = V$ and all vertices have been processed.

Input: $G = (V, E)$ a graph (can be directed), a vertex $s \in V$ and a non-negative weight function $w : E \to \mathbb{R}^+ \cup \{0\}$.

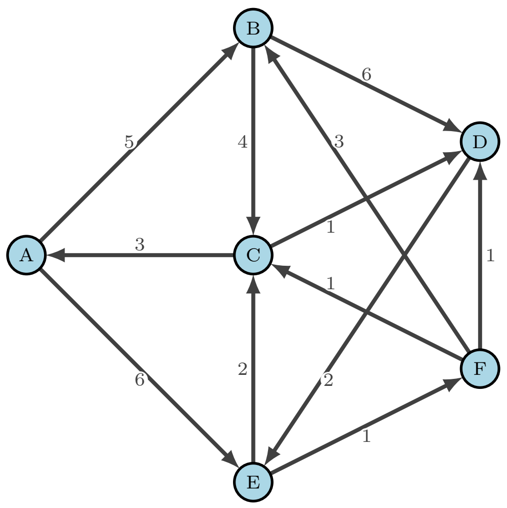
```
1.  $t(s) \leftarrow 0$; for all $v \in V, v \neq s$, if $(s, v) \in E(G)$, $t(v) \leftarrow w((s, v))$, otherwise, $t(v) \leftarrow \infty$
2.  for all $v \in V, v \neq s$, if $(s, v) \in E(G)$, $p(v) \leftarrow s$, otherwise, $p(v) \leftarrow *$
3.  $K \leftarrow \{s\}$
4.  loop 1: while $K \neq V$
5.      let $v$ be a vertex in $V \setminus K$ with minimum $t(v)$ value.
6.      loop 2: for every edge $e = (v, u), u \notin K$
7.          if $t(u) > t(v) + w(e)$, then:
8.              $t(u) \leftarrow t(v) + w(e)$
9.              $p(u) \leftarrow v$
10.     end loop 2
11.     $K \leftarrow K \cup \{v\}$
12. end loop 1
```

Here edges have been written as (u, v). If Dijkstra's algorithm is being used on a directed graph, then this edge should be understood as $\overrightarrow{uv}$. The key thing to note about the algorithm is that for every vertex added to K, it updates the distances to all vertices with an incoming edge from this vertex.

Let $n = |V(G)|$ and $m = |E(G)$. Then, the Dijkstra's algorithm takes at most $c \cdot n^2$ steps for some constant c. Here we assume that finding the minimum of the list $t(v)$ at every stage takes $c'n$ steps and since we repeat the cycle n times, we get the previously stated running time. We also note that every edge in the graph is relaxed (updating the $t(v)$ value across an edge) at most once, and since number of edges is bounded by n^2, so this running time takes into account that too.

We also note that with better data structures (heaps) to determine the minimum of $t(v)$'s, we can achieve improvements in the running time of the Dijkstra's algorithm.

For an example, consider the following graph and the problem of finding shortest paths from A:



The following table shows the steps of Dijkstra's algorithm. The leftmost column shows the new vertex added to K at every turn. Also the colored distance show final distance which are not going to change since the vertex has been added to K.

K	$S=A$	B	C	D	E	F	$p(B)$	$p(C)$	$p(D)$	$p(E)$	$p(F)$
A	0	5	∞	∞	6	∞	A	*	*	A	*
B	0	5	9	11	6	∞	A	B	B	A	*
E	0	5	8	11	6	7	A	E	B	A	E
F	0	5	8	8	6	7	A	E	F	A	E

Since C and D both have $t(v)$ as 8, their shortest distance cannot change further. So the algorithm can stop here.

We need the following claim to prove the correctness of the Dijkstra's algorithm. Notice that we prove something stronger, namely that once a vertex is added to K, its shortest distance has been discovered, and for every vertex outside, $t(v)$ maintains the shortest distance using only vertices in K.

Claim: At every stage of the running of the Dijkstra's algorithm, the following two statements are true:

1. $\forall u \in K$, $t(u) = d(s,u)$
2. $\forall u \notin K$, $t(u)$ is the shortest distance of a s,u path in the subgraph induced on the vertices $K \cup \{u\}$.

Proof: We will use the notation that for any two vertices, if uv is an edge in the graph, then $w(uv)$ is the weight of the edge as defined by the weight function, but if there is no such edge in the graph, then we define $w(uv) = \infty$.
We will use induction on the size of K. We initialize with $K = \{s\}$, then $d(s,s) = t(s) = 0$ and for $u \neq s$, $t(u) = w(su)$. These values satisfy both the above conditions.
Let $K \neq V(G)$ and let the claim be true for K. Now, we suppose that the algorithm picks a new vertex $v \notin K$ and we add it to K. Then, we need to check that the claim still holds.
For the first statement, we need to check that $t(v) = d(s,v)$. For this, we note that $t(u)$ is (by induction) the shortest path to any vertex using only vertices in $K \cup \{u\}$. And v is the vertex with the smallest $t()$ value from the vertices in $V(G)\setminus K$. Since any path to v must exit K, the shortest s,v-path must directly exit K to v, since if it exits K to another vertex u, then the path to v will be longer. So $t(v) = d(s,v)$.
For seeing that the second statement also holds, we need to check that on the graph induced on $K\cup\{u,v\}$, for any vertex $u \notin K$ and $u \neq v$, the updated $t(u)$ is the shortest path. We notice that the way we update $t(u)$ is , $t(u) = \min(t(u), t(v) + w(vu))$. Since the only new path in this induced subgraph can be through v and we are updating t as the shorter of these two paths, we are making sure that $t(u)$ remains the shortest path in $K \cup \{u,v\}$.

From the claim we can see that when the algorithm ends with $K = V(G)$, we will have found the shortest distances to all vertices.

Bellman-Ford Algorithm

Conservative weight function: Given a directed graph G, an edge weight function $w : E(G) \to \mathbb{R}$ is said to be **conservative** if there are no directed cycles with a negative total weight.

Theorem: Given a directed graph G with a conservative edge weight function $w : E(G) \to \mathbb{R}$, if there exists a directed s, v-walk with k edges and total weight t, then there exists a directed s, v-path with at most k edges and total weight at most t.
I didn't have time in class to do this proof! Recall our proof from Lecture 3 for the statement that every s, v-walk contains a s, v-path: we found the path by deleting the walk between the first and last occurance of a vertex that is repeated, and repeating this until no vertex is repeated in the walk. This argument works here as well and will give us the path with at most k edges. But the catch here is that in this case we cannot conclude that the total weight is also less than t because we are deleting some walk and it may have negative total weight. So to fix this proof, we need to ensure that we delete a cycle every time, and since the flow is conservative, we will be reducing the total weight because there are no negative weight cycles. For this we simply modify proof by saying that if the s, v-walk is not a path, then some vertex is repeated in it. Of all such repeats, pick the repeat where the number of edges between the vertex and its repeat is the smallest. We notice that this has to be a cycle and we can delete it.

Theorem: Given a directed graph G and a conservative edge weight function, if P is the shortest s, v-path and u is another vertex on this path, then the part of this path between s and u is a shortest path between them.
I didn't have time in class to do this proof!

Definition: $t_k(v)$ is the length of the shortest s, v-path using at most k edges.

Theorem: Let G be a directed graph with a conservative edge weight function $w : E(G) \to \mathbb{R}$. Let $s, v \in V(G), s \neq v$. Then for every $k \geq 1$,
$t_k(v) = \min\{t_{k-1}(u) + w(e) : e = (u, v), e \in E(G)$.
Note: if this set is empty, then there is no path reaching v and we set $t_k(v)$ to be ∞.

This gives us the Bellman-Ford algorithm for finding shortest paths from s.

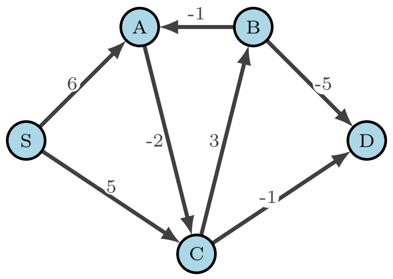
Input: $G = (V, E)$ a directed graph, a vertex $s \in V$ and a weight function $w : E \to \mathbb{R}$.

```
1.  $t_0(s) \leftarrow 0$; for all $v \in V, v \neq s$, $t_0(v) \leftarrow \infty$
2.  for all $v \in V$, $v \neq s$, $p_0(v) \leftarrow *$
3.  **loop 1:** for $k$ from 1 to $n - 1$:
4.      $t_k(s) \leftarrow 0$
5.      **loop 2:** for every vertex $v \in V, v \neq s$:
6.          $t_k(v) \leftarrow t_{k-1}(v)$
7.          $p_k(v) \leftarrow p_{k-1}(v)$
8.          **loop 3:** for every incoming edge $e = \overrightarrow{uv}$:
9.              **if** $t_k(v) > t_{k-1}(u) + w(e)$, then:
10.                 $t_k(v) \leftarrow t_{k-1}(u) + w(e)$
11.                 $p_k(v) \leftarrow u$
```

```
12.         end loop 3
13.     end loop 2
14. end loop 1
15. for every vertex $v \in V, v \neq s$, $d(s,v) \leftarrow t_{n-1}(v)$, $p(v) \leftarrow p_{n-1}(v)$
```

The Bellman-Ford algorithm takes at most $c \cdot n \cdot m$ steps for some constant c and where $n = |V(G)|$ and $m = |E(G)$.
We can see this because the relaxing of all edges (updating the t values across an edge) happens n times.

We didn't cover this in class: The above algorithm can be modified to show if the weight function is conservative. Then loop 1 runs until n and we check if the values $t_{n-1}(v) = t_n(v)$ for all $v \in V$. If yes, then the weight function is conservative, otherwise not.
Consider the following graph:



We can show the steps of the Bellman-Ford algorithm in the table below. Here k is from $t_k()$, the algorithm proceed with find t_0 then t_1 and so on by relaxing all the edges of the graph with every iteration. This we do so by fixing the order of vertices for every round. Here the order in which we process vertices is S, A, B, C, D and this order is fixed in every iteration (every k).

k	S	A	B	C	D	$p(A)$	$p(B)$	$p(C)$	$p(D)$
0	0	∞	∞	∞	∞	*	*	*	*
1	0	6	∞	5	∞	S	*	S	*
2	0	6	8	4	4	S	C	A	C
3	0	6	7	4	3	S	C	A	B
4	0	6	7	4	2	S	C	A	B

Lecture 13

Depth First Search, Topological ordering in DAGs.

DFS

DFS. Input: G and s (starting vertex). Important lists in the table:

- $d(v)$: depth of the vertex v. Initially, $d(s) = 1$ and $d(v) = *, \forall v \neq s$. This denotes the order in which the vertices are visited, so this list generates a **preordering** of the vertices.
- $f(v)$: finishing index of the vertex v, also called a **postordering** of the vertices.

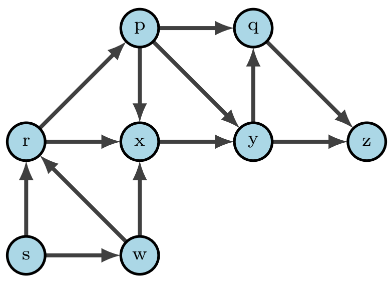
- $p(v)$: parent of v in the DFS tree
- a: current active vertex
- D: the maximum depth encountered so far
- F: the maximum of finishing indices so far

Input: $G = (V, E)$ and a vertex $s \in V$.

```
1.  $d(s) \leftarrow 1$; for all other vertices $v \in V, v \neq s$, $d(v) \leftarrow *$
2.  for every vertex, $f(v) \leftarrow *$ and $p(v) \leftarrow *$
3.  $D \leftarrow 1$, $F \leftarrow 0$, $a \leftarrow s$
4.  **loop**
5.      **if** there is an edge $e = \overrightarrow{av}$ such that $d(v) = *$, **then:**
6.          $D \leftarrow D + 1$
7.          $d(v) \leftarrow D$
8.          $p(v) \leftarrow a$
9.          $a \leftarrow v$
10.     **otherwise:**
11.         $F \leftarrow F + 1$
12.         $f(a) \leftarrow F$
13.         **if** $p(a) \neq *$, **then:**
14.             $a \leftarrow p(a)$
15.         **otherwise:**
16.             **if** there is a vertex $v$ such that $d(v) = *$, **then:**
17.                 let $a$ be such a vertex $v$
18.                 $D \leftarrow D + 1$
19.                 $d(a) \leftarrow D$
20.             **otherwise:**
21.                 **stop:**
22. **end loop**
```

The DFS algorithm takes $c(n + m)$ steps where $n = |V(G)|$ and $m = |E(G)|$. Also, if there are vertices with no directed path from s, then we will get a forest after running the algorithm.

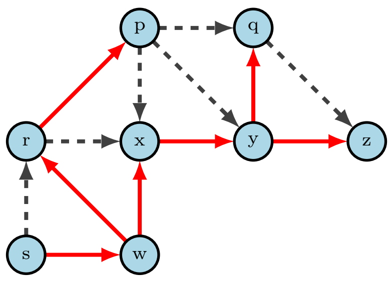
Consider the following graph:



The DFS algorithm when run on the above graph will give the following results:

v	s	p	q	r	x	y	z	w
$d(v)$	1	8	6	7	3	4	5	2
$f(v)$	8	5	2	6	4	3	1	7
$p(v)$	*	r	y	w	w	x	y	s

And the DFS tree will look like:



Lemma: During the running of the DFS algorithm, let u be a vertex that becomes active, that is, $d(u) = *$ and $a \leftarrow u$. Let $X \subseteq V(G)$ of all vertices with $d() = *$. Also let $S \subseteq X$ be the set of all vertices with a directed path from u in the subgraph induced by X. Then for every $v \in S$, u will be an ancestor of v in the DFS tree.
Proof: we can see this with induction on the size of S. If $S = \phi$, then this is trivially true. Now let S be non empty. The depth first search algorithm will pick a vertex v from S and $a \leftarrow v$. The set of unexplored vertices that are reachable from v, call S' will be a strict subset of S, $S' \subset S$, hence by induction, v will be an ancestor of all such vertices and consequently u will also be an ancestor. The algorithm will return to u only after processing of v is finished and a new neighbor of u is made active.When the processing of v is finished, we notice that the vertices of S which don't have a depth index are precisely the vertices which didn't have a directed path from v. So all these vertices will still have a directed path from u using only vertices with $d() = *$. So, we repeat can repeat this process of picking another neighbor of u until all the vertices in S become its descendents.

After running the DFS algorithm, the edges in the graph G can be categorized as follows:

- Tree edges - edges that belong to the tree
- Forward edges - edges $\overrightarrow{uv}$ such that u is an ancestor of v in the DFS tree, that is, there is a directed path from u to v in the DFS tree.
- Back edges - edges $\overrightarrow{uv}$ such that v is an ancestor of u in the DFS tree, that is, there is a directed path from v to u in the DFS tree.
- Cross edges - edges $\overrightarrow{uv}$ such that there is no directed path in the DFS tree from u to v or from v to u.

Lemma: We make the following observations for all edges $\overrightarrow{uv}$ in G:

- If its a tree edge, then $d(u) < d(v)$ and $f(u) > f(v)$.
- If its a forward edge, then $d(u) < d(v)$ and $f(u) > f(v)$.

- If its a back edge, then $d(u) > d(v)$ and $f(v) > f(u)$.
- If its a cross edge, then $d(u) > d(v)$ and $f(u) > f(v)$.

If $\overrightarrow{uv}$ is a tree edge, then u is the parent, and so $d(u) < d(v)$. Since the only backtracking happens when a vertex is finished and then the active vertex goes to the parent, so a child always finishes before the parent. So $f(u) > f(v)$.
If $\overrightarrow{uv}$ is a forward edge, then u is an ancestor of v. Then there are vertices $u = u_1, u_2, .., u_k = v$ such that u_i is the parent of u_{i+1}. Then since $d(u_i) < d(u_{i+1})$ and $f(u_i) > f(u_{i+1})$, so the result follows.
If $\overrightarrow{uv}$ is a backward edge, then v is an ancestor of u. The result follows from similar reasoning as for a forward edge.
If $\overrightarrow{uv}$ is a cross edge, then first we note that $d(u)$ cannot be less than $d(v)$. Because otherwise, since there is a edge from u to v, v will be a descendent of u by the Lemma above. So we conclude that $d(u) > d(v)$. If $f(v) = *$ when u becomes active, then v will be an ancestor of u and we have a directed path from v to u in the DFS tree. So, $f(v) < f(u)$.

We note briefly here that DFS algorithm can also be run on undirected graphs. We can observe that by introducting two directed edges $\overrightarrow{uv}$ and $\overrightarrow{vu}$ for every undirected edge (u, v).

Topological ordering

Given a Directed Acyclic Graph (DAG for short), an ordering of the vertices $(v_1, v_2, ..., v_n)$ is said to be a topological ordering if for every edge $\overrightarrow{v_iv_j}$, $i < j$. In particular, there are two vertices s (source) and t (sink) which have only outgoing and incoming edges respectively.

Theorem: A directed graph G has a topological ordering of the vertices if and only if it is acyclic.
Proof sketch: If a directed graph has a topological ordering, say f, then since for every edge $\overrightarrow{uv}$, $f(u) < f(v)$, it cannot have a directed cycle.
On the other hand, if the graph does not have any cycles, then we notice that there must be a vertex which has no incoming edges. Otherwise, we start at any vertex v, and pick a vertex from which it has an incoming edge, and we repeat this until a vertex repeats on this path, thereby giving us a cycle. Let us call this vertex with no oncoming edges as v_1. We notice that $G \setminus \{v_1\}$ is also a directed acyclic graph, and so we can repeat this process and pick another vertex here that has no incoming edges. We will call this vertex v_2. We continue this process until we get the required ordering $v_1, v_2, ..., v_n$. We notice that this is a topological ordering of the graph because at any stage i, v_i was a source in the graph $G \setminus \{v_1, ..., v_{i-1}\}$ so all edges outgoing from v_i must be to $v_j, j > i$.

We use a topological ordering to give a much faster algorithm for finding the shortest and longest paths from the source vertex s.

Input: DAG $G = (V, E)$ with a topological ordering $(s = v_1, v_2, ..., v_n)$ and an edge weight function $w : E(G) \to \mathbb{R}$.

```
1. $t(v_1) \leftarrow 0$; for $i > 1$, $t(v_i) \leftarrow \infty$
2. **loop:** for $i$ from 2 to $n$
3.     $t(v_i) \leftarrow \min\{t(v_j) + w(e) : e = \overrightarrow{v_jv_i}\}$
4. **end loop**
```

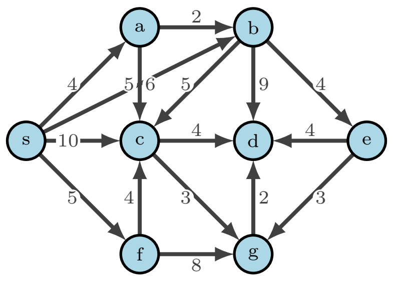
Notice: if we replace min with max in the algorithm above, then the above algorithm will also compute the longest paths from $s = v_1$.

The above algorithm finds all shortest paths from v_1 in $c \cdot (n + m)$ steps, where $n = |V(G)|$ and $m = |E(G)|$.

We also observe that the DFS algorithm can decide if a graph G is acyclic and if yes, provide us with a topological ordering.

Theorem: Consider a directed graph G with the DFS algorithm run from vertex s. G is acyclic if and only if the DFS algorithm does not discover a back-edge. Also, if there were no back-edges, then the reverse of postordering of the vertices (the vertices ordered with decreasing $f(v)$ value) gives a topological ordering of the vertices.
Proof sketch: it follows from the properties of all the types of edges discussed earlier, only back edges form directed cycles. Also, only for back edges $\overrightarrow{uv}$, $f(u) < f(v)$. For every other kind of edge, $f(u) > f(v)$.

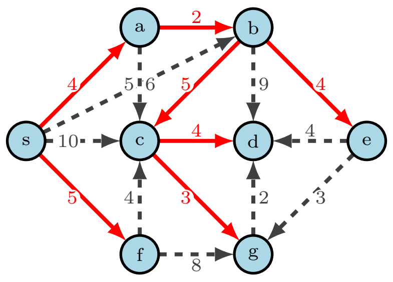
Consider the following graph:



The DFS algorithm when run on the above graph will give the following results:

v	s	a	b	c	d	e	f	g
$d(v)$	1	2	3	4	5	7	8	6
$f(v)$	8	6	5	3	1	4	7	2
$p(v)$	*	s	a	b	c	b	s	c

And we will get the following tree:



Let $t(v)$ denote the minimum distance and $T(v)$ the maximum. The topological order of vertices is s, f, a, b, e, c, g, d. Then we get the following computations for the algorithm:

v	s	f	a	b	e	c	g	d
$t(v)$	0	5	4	5	9	9	12	13
$T(v)$	0	5	4	6	10	11	14	16