Lecture Notes

CS254 - Algorithmic Graph Theory

General

1. V: Vertices(Nodes), E: Edges(Pairs of Nodes)
2. Pairs of nodes comprising relation E are called *edges*
3. Two nodes connected by an edge are called *adjacent*
4. $G = (V, E)$: Graph with sets of nodes V and edges E
5. Graph (G) "on" V is an Irreflexive, symmeric relation defined by $E = R_{\rightharpoonup} : V \leftrightarrow V$ (V is any finite set)
6. **Empty graph** has no edges: $(V, \varnothing)$
7. **Complete graph** $K(n)$ contains all possible edges: $K(V) = (V, E)$, where $E = \{(u, v) \in V^2 | u \neq v\}$
8. Graph G is **bipartite** or *two-coloured* if set of nodes can be partitioned into 2 disjoint subsets $V = V_1 \cup V_2$ s.t. every edge in E connects 2 nodes from diff. subsets V_1, V_2 are *colour classes*
9. $K(n)$ can be read as: 'Any graph isomorphic to $K(\mathbb{N}_n)$'
10. $K(V_1, V_2) = (V_1 \cup V_2, (V_1 \times V_2) \cup (V_2 \times V_1))$ is called a complete bipartite graph. $K(m \in \mathbb{N}, n \in \mathbb{N})$ - any graph isomorphic to $K(H, W)$ with m houses, n wells
11. A graph with k colour classes is called k-partite
12. Complete graph has $\frac{n(n-1)}{2}$ edges.
13. Connected graph stays conn. when adding edges
14. Acyclic graph stays acyclic when removing edges
15. Trees are maximal among acyclic graphs

---

1. **Eulerian** cycle visits each edge only once.
2. **Hamiltonian** cycle visits each node only once.
3. $V \neq \varnothing, |V|$ finite.
4. Directed graph: ordered pairs: $e = (v, w) \in E$
5. Undirected: unordered $e = \{v, w\} \in E$
6. Self-loops: $e = (v, v)$
7. A graph is **simple** if no loops and multiple edges.
8. edges(e)=v(source) w(destination in dir.)$G \in E$
9. *Multiplicity*: number of edges between 2 nodes.
10. *Adjacent* nodes: Nodes, connected by an edge.
11. *Incident* nodes: Nodes that an edge connects.
12. Self-loops count twice in *Vertex degree*
13. in-deg(v) $\stackrel{\text{def}}{=}$ num. edges where v is destination.
14. out-deg(v) $\stackrel{\text{def}}{=}$ num. edges where v is source.
15. $v \to w : vw \in E$
16. $v \to^* w : \exists v \looparrowright w$ or w is reachable from v.
17. Graph G is Eulerian if it has an Eulerian cycle.
18. G' is a *subgraph* of $G(G' \subseteq G)$ if $V' \subseteq V, E' \subseteq E$
19. G' **spanning subgraph** of $G(G' \sqsubseteq G)$ if $V' = V, E' \subseteq E$
20. $R_{\subseteq}, R_{\sqsubseteq} : \mathcal{G}(V) \leftrightarrow \mathcal{G}(V)$ are part. orders on $\mathcal{G}(V)$
21. **Tree**: connected, acyclic graph.
22. **Forest**: acyclic graph (not necessarily connected)
23. If $\deg(v) = 1$ in a tree, then v is a *leaf*

Graph Connectivity

1. **Walk** (of len. k) is a sequence $(u, u_1, ...u_{k-1}, v)$ s.t. every two consecutive nodes in the sequence are connected by an edge:$(u \rightharpoonup u_1) \wedge ... \wedge (u_{k-1} \rightharpoonup v)$
2. $u \looparrowright v$ "nodes u and v are connected by a walk"
3. **Tour** is a walk that returns to the starting node
4. Nodes u, v in a graph are *connected*, if $\exists u \looparrowright v$.
5. A graph is connected if all (u, v) are connected.
6. Connectivity is equivalence relation on the set of all nodes in a graph: $R_{\looparrowright} : V \leftrightarrow V$
7. Equivalence classes of $R_{\looparrowright}$ are "connected components" of graph G. A graph is connected iff it has only 1 connected component.
8. A walk where all E are distinct is a **Path**: $u \rightsquigarrow v$ $u = u_0 \rightharpoonup ... \rightharpoonup u_k \rightharpoonup v \ \forall i, j \in \mathbb{N}_{k+1} | u_i \neq u_j$
9. **Cycle** is a tour with no edges repeated.
10. A graph without cycles is called *acyclic*
11. $R_{\rightsquigarrow} : V \leftrightarrow V$ is equivalence relation
12. $\deg(v) = |\{u \in V | v \rightharpoonup u\}$ (num adjacent nodes)
13. **Simple Path** is a walk that repeats no vertices.
14. **Simple Cycle**: tour with no vertices repeated except $v_0 = v_n$
15. A graph is planar, if can be embedded in the plane s.t. the lines representing different edges do not cross.
16. Usually want to identify graphs which are "the same up to a renaming of nodes"
17. Graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ are *isomorphic* if bijective func. $f : V_1 \leftrightarrow V_2$, preserving edges exists: $\forall u, v \in V_1 | (u, v) \in E_1 \equiv (f(u), f(v)) \in E_2$, where bijective f is an isomorphism between G_1 and G_2
18. **Subdivision** of a graph $G = (V, E)$: choose an edge $\{u, v\} \in E$, remove it, add a new vertex $x \notin V$, and insert the edges $\{u, x\}$ and $\{x, v\}$.
19. **Undirected:** $\{x, y\} \in E$, **Directed:** $(x, y) \in E$.
20. **Independent Set** $X \subseteq V$ if $\forall x, y \in X : \{x, y\} \notin E$.

Theorems

1. $\forall u, v \in V | \exists u \rightsquigarrow v$ iff $\exists u \looparrowright v$
2. G has Euler tour iff: G connected, $\forall v \in V : |v|$ even (Euler, Hierholzer)
3. **Handshaking Lemma**: $\sum_{v \in V} \deg(v) = 2 \cdot |E|$
4. Let $G = (V, E)$ be a tree. Then $|V| = |E| + 1$
5. Every tree with at least one edge has a leaf.
6. Partial order $R_{\sqsubseteq}$ on set of all asyclic graphs on finite set V. Graph $G = (V, E)$ is maximal iff it's a tree.
7. Graph is **planar** iff it has no subgraph isomorphic to a graph obtained from $K(5)$ or $K(3, 3)$ by a sequence of subdivisions.

Intro

1. For graph $G = (V, E)$: **Order** $n = |V|$, **Size** $m = |E|$.
2. $\text{Deg}(v) = \text{in-deg}(v) + \text{out-deg}(v)$. $\sum_{v \in V} \deg(v) = 2|E|$. Δ denotes the maximum degree $d(v)$ of a node in $v \in V$.
3. **Subgraph** $G'=(V', E')$ of $G=(V, E)$ if $V' \subseteq V \wedge E' \subseteq E$. **Spanning** subgraph if $V' = V$ (share all vertices). **Induced** subgraph if $\forall e_j \in E$ incident on $v'_i \in V' : e \in E'$ (some nodes but all original edges connecting them).
4. **Multigraph** (pseudograph) allows multiple edges between two vertices and self-loops, so E becomes a multiset of edges, each being a multiset of vertices over V. Otherwise the graph is **simple**.
5. **Forest**: undirected graph without a cycle. **Tree** is a connected forest. **Spanning tree** if the spanning subgraph is a tree. **DAG** is a directed asyclic graph.
6. Graph directed: $0 \le m \le 2 \cdot \binom{n}{2}$; Forest: $m \le n - 1$; Simple undirected: $0 \le m \le \binom{n}{2}$; DAG: $m \le \binom{n}{2}$
7. **Incident** edge: touches a vertex. **Endpoints** u, v of $e = (u, v) \in E$ **Adjacent vertices/edges:** share an edge/vertex. **Directed edge:** Goes from *tail* to *head* $(u \to v)$. **Consecutive edges:** $(u, x), (x, v)$ $(\text{tail}_1 = \text{head}_2)$ **Consecutive nodes:** Connected as *tail* $\to$ *head*.
8. **Path** $P = v_1, .., v_k$ is a $v_1 \rightsquigarrow v_k$ or (v_1, v_k)-path. Undirected G **connected** if $\forall u, v \in V : \exists (u \rightsquigarrow v)$ path. Directed G **connected** if $\forall u, v \in V : \exists (u \rightsquigarrow v) \vee (v \rightsquigarrow u)$, **strongly connected** if $\forall u, v \in V : \exists (u \rightsquigarrow v) \wedge (v \rightsquigarrow u)$
9. $$f \in O(g) \text{ iff } \begin{cases} \forall n \in \mathbb{N}. \exists c > 0 : f(n) \le c \cdot g(n) \\ \lim_{n \to \infty} \frac{f(n)}{g(n)} < +\infty \end{cases}$$ $$f \in \Theta(g) \text{ iff } f \in O(g) \wedge g \in O(f)$$ $$f \in \Omega(g) \text{ iff } g \in O(f)$$ $$f \in \omega(g) \text{ iff } \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$
10. Can represent a graph using
    - **Edge list**: directed/undirected depends on interpretation. Storage $O(m)$, useful for I/O, but not many other operations. Computing $d(v)$ takes $O(m)$.
    - ***Adjacency list (default)***: takes $O(n + m)$ storage, but checking if $(u, v) \in E$ takes $\Theta(\min\{d(v), d(u)\})$.
    - ***Adjacency matrix***: Storage $O(n^2)$, wasteful since most graphs are sparse with $m \ll n^2$. Find 1 neighbour in $O(1)$, all in $\Theta(n)$, most other operations in $\Omega(n^2)$.
    - **Implicit representation**: e.g. edge exists from x to y if y is in the ball with radius $r(x)$ around x.

Bipartiteness

1. Undirected $G = (V, E)$ is **bipartite** if $V = V_1 \cup V_2$ with one endpoint per edge: $\forall e \in E : |e \cap V_i| = 1$ for $i = 1, 2$. Directed bipartite if $V = V_1 \cup V_2$, $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$.
2. **Corollary**: graph is bipartite iff it has no odd-length cycle. Bipartite graph has 2 independent vertex sets.
3. Let $\sim$ denote **connectedness-relation** on G: $u \sim v \equiv u$ connected to $v \in V$. Reflexive, symmetric, transitive. Its equivalence classes are G's **connected components**

Simple explore

```
def Simple-Explore(G,s):
  R = {s}; # BFS if R is queue, DFS if R is stack
  while ∃{u,v} ∈ E s.t. u ∈ R ∧ v ∉ R:
    R = R ∪ {v};
```

resulting R contains connected component of G which is a tree with shortest (s, v)-path from root s to all $v \in V$ if BFS (R is a queue q, $u = q[0]$), DFS if R is a stack.

4. **Lemma**: if G bipartite, then $\nexists\{u, v\}$ edge with u, v on the same level i in BFS-tree because otherwise the lowest common ancestor L_{i-t}, t layers above L_i produces a cycle of length $2t + 1$, which is odd, so contradiction.

Bipartiteness BFS $\quad O(n + m)$

```
def TestBipartiteness(G):
  for each unvisited node s in G:
    BFS(s), assigning levels; #connected component
    if ∃{u,v} ∈ E with level(u) = level(v):
    return false;
  return true;
```

BFS

1. **BFS Property 1**: edges not within BFS tree can only connect successive layers or vertices on the same layer. **BFS property 2**: BFS tree contains shortest (s, v)-path for every vertex v reachable from s.
2. BFS helps find (strongly) connected components, test bipartiteness and sort topologically, all running in $O(1) + \sum_{u \in V} (d(u) + O(1)) = \underline{O(n + m)}$ time if adjacency list.
3. Adj list: go through all V in $O(n)$, for vertex v go through neighbours in $O(d(v))$, but checking if $(u, v) \in E$ takes $\Theta(\min(d(u), d(v)))$.
4. **Sink** is a vertex $u \in V$ if $\deg_{in}(u) = n - 1, \deg_{out} = 0$. *No simple directed graph has more than one sink.* If $(u, v) \in E$ then u isn't a sink, if false, then v isn't one. Repeat $n - 1$ times until 1 left, to check if it's a sink, taking $O(3(n - 1))$ with adjacency matrix.
5. **Theorem**: any algorithm determining if graph is bipartite that has input as undirected graph $G = (V, E)$ represented as an $n \times n$ adj matrix has $\Omega(n^2)$ runtime. **Proof**: consider ALG on a **star** $G_0 = (V, E_0)$ with $V = \{1, 2, ..n\}$, $E_0 = \{\{1, i\} : 2 \le i \le n\}$. Suppose checked $< \binom{n-1}{2}$ queries, then there are some entries ALG hasn't visited - editing those constructs G^* s.t. G and G^* are **indistinguishable** by ALG, so both return same result, but shouldn't, hence must visit n^2 items.

DFS numbering

1. Find **connected components** by running DFS or BFS from vertex $s \in V$, removing that connected component, repeat until $V = \varnothing$. If component C has n_C vertices and m_C edges, either take $O(n_c + m_c)$ and $O(n+m)$ in total.
2. **DFS number** $N[v]$: distinct DFS finish time at $v \in V$, if u ancestor of v (*forward edge*) or not related (*cross edge*) $N[u] > N[v]$, else *backward edge*: $N[u] < N[v]$.

DFS numbering $\mathbf{O}(n+m)$

```
c = 0; ∀v ∈ V : N[v] = 0; # counter, dfs numbering
while ∃v ∈ V : N[v] = 0 do DFS(G,v);
DFS(G,x): # recursive DFS algorithm
  visit x;
  for x,y∈ E do
    if N[y] = 0 then DFS(G,y);
    T = T ∪ {{x,y}}; # DFS tree edges (global)
  N[x] = ++c; # backtracking increments dfs number
```

Maintain non-active, active & finished nodes.

3. **DFS-tree** in 1 execution of the recursion. **DFS-forest**: sum of disjoint trees. **Forward edge** from node to its descendant. **Backward edge** from node to its ancestor. **Cross edge** between non-ancestor/descendant edges.
4. DFS trees only have forward/backward, no cross edges.

Directed Cycles, Topological Sort

1. Asyclic graph with no parallel edges has $0 \le m \le n-1$ edges if undirected, and $0 \le m \le \binom{n}{2}$ if directed.
2. **Directed Acyclic Graph (DAG)** is a directed graph $G = (V, E)$ with no cycles.
3. **Topological sort** is map $\phi : V \to \{1, .., n\}$ s.t. $\forall (u, v) \in E : \phi(u) < \phi(v)$ - visit order in digraph. In DAG, DFS numbering would produce $N[u] < N[v]$ (since no backwards edges), so its reverse is topologically sorted.
4. **Theorem**: digraph has topological sort iff it's a DAG. **Proof**: $(\Rightarrow)$ suppose G has topological sort, FTSOC assume G has cycle $C = \{x_0, ..x_k\}$. wlog let $x_0 = \min_{i \in \{1,..k\}}\{\phi(x_i)\}$, or the vertex of C with smallest number. Then $(x_k, x_0) \in E$ but $\phi(x_k) \ge \phi(x_0)$
$(\Leftarrow)$ Suppose G is DAG, then $\exists$ a sink, number it n and delete from G, recursively number the rest. Find a sink in O(n), decrement out-deg of each node directed at v, so for each node u takes O(out-deg(u)), overall O($n+m$).
5. **Reminder:** if $v \in V$ is a **sink**, then all non-sink $u \in V$ have to have an edge $(u, v) \in E$. Can have ≤ 1 sink.
6. **DAG cycle detection**: compute DFS numbering N on digraph G, set $\phi(v) = n - N[v] + 1$. For every $(u, v) \in E$ check if $\phi(u) < \phi(v)$, if some edge fails then G is not DAG, else it is DAG, hence has no cycles.

Connected Components

1. **Reminder**: for undirected graph $G = (V, E)$, vertices $u, v \in V$ are **connected** if $\exists$ path $u \rightsquigarrow v$ and **strongly connected** if also $\exists$ path $v \rightsquigarrow u$.
2. Connectedness relation an equivalence relation - its equivalence classes are **(strongly) connected components (S)CC** of G. Can find SCC/CCs with BFS/DFS in $O(n+m)$ time.
3. Can use SCC to design efficient algorithms. Let $G = (V, E)$ be digraph. Find SCCs of G, ***contract*** each SCC into a single **supervertex**. Becomes **meta graph** G^* once all are contracted. **Claim**: contracted G^* is asyclic
Proof: FTSOC suppose G^* not asyclic, then $\exists$ cycle $x_0, ..x_k, x_0$ s.t. each x_i is a SCC. But then all vertices in any x_i, x_j are strongly connected: use path $x_i, x_{i+1}, ..x_j$ in one direction, and path $x_j, x_{j+1}, ..x_0, x_1, ..x_j$ in other. Contradiction - SCCs are not distinct, so G^* is DAG.
4. **SCC algorithm recipe**: 1. Find SCCs, contract each. 2. topologically sort contracted components. 3. Use DP on DAGs to solve the problem.
5. **Observation**: if we start DFS at any $x \in V$ belonging to sink of G^* meta graph, then we explore the whole SCC of x, but no other super vertex.
6. **Lemma**: let C_1, C_2 be two SCCs of G with $(C1, C2)$ edge in meta graph: $\max_{u \in C_1} N(u) > \max_{v \in C_2} N(v)$ as $(u, v) \in E, u \in C_1, v \in C_2$, so (u, v) is a cross edge, so $N(u) > N(v)$.
Corollary: vertex with maximal DFS number is contained in source vertex of meta-graph.
7. **Reverse graph**: let $G' = (V, E')$ be reverse graph of $G = (V, E)$ with all edges reversed. Then meta graph of G' is the reverse of meta graph G^* of G, so same SCCs.
Corollary: if run DFS on G', then vertex with maximal DFS number is contained in sink SCC of G^*
8. **Kosaraju's Algorithm**: find SCCs via DFS. 1. Compute reverse graph G' of G. 2. Run DFS on G' and compute finish number $N(v)$ for every vertex $v \in V$. 3. Run DFS on G where restarting is always done at v with **max-value** of $N(v)$ among yet unvisited vertices.
9. **Theorem**: Kosaraju's algorithm finds all SCCs of a digraph in time $O(n+m)$.

Biconnected components and DFS

1. Undir G is **biconnected** if $G \setminus \{v\}$ is connected $\forall v \in V$.
2. **Cut-vertex** or **articulation point** is vertex whose removal disconnects the graph.
3. **Biconnected component** of G is its maximal biconnected subgraph.

4. Graph is **k-vertex connected** if removal of any $k-1$ vertices leaves remaining graph connected.

5. Let $\sim$ be equivalence **relation** on E s.t. $e_1 \sim e_2$ iff e_1, e_2 are contained in a simple cycle in G or $e_1 = e_2$. It's reflexive, symmetric and transitive.

6. **Transitivity**($\sim$):if $\exists$ two simple cycles containing e_1, e_2, and e_2, e_3 respectively, then $\exists$ one containing e_1, e_3. **Proof:** find simple cycle as follows. Let e_2 together with $e_1 = (u_0, u_1), e_3 = (v_0, v_1)$ be contained within simple cycles $C_1 = (u_0, u_1, ..u_k)$ and $C_2 = (v_0, v_1, ..v_r)$, respectively. Let $s \in \{1, ..k-1\}$ and $l \in \{2, ..k\}$ be smallest and largest index s.t. $u_s, u_l \in C_2$. Assume their indices in C_2 are s' and l', which must exist, be different; if $s' < l'$ then sequence $u_0, ..u_s{=}v_{s'}, v_{s'-1}, ..v_0, v_r, ..v_{l'+1}, v_{l'}{=}u_l, ..u_k)$ is a simple cycle, similarly for $l' < s'$ $\square$

7. If root node has at least two children then it will always be an **articulation point**, otherwise isn't. Leaves are never articulation points. Internal node is articulation point when none of its descendants have a back edge to one of its proper ancestors.

8. **Low-point** of node v in DFS-tree is lowest level (closest to the root) among the neighbours of (all) nodes in the subtree T_v rooted at v (subgraph starting at v).

9. **Articulation Point search**: compute DFS tree. Compute level and low-point for each node. $\forall$ internal nodes check if low-point of one of its children is $\geq v$ if so, it's articulation, else not. Can do using one DFS traversal.

Articulation point search algorithm$O(n+m)$

```
def Articulation-Points(G,x,l):
  x.level = l; x.visited = true;
  ∀ {x,y} ∈ E do: # visit children
    if y unvisited: Articulatoin-points(G,y,l+1);
  x.low_point = x.level; x.articulation = false;
  ∀ {x,y} ∈ E do:
    if y.level = l+1: # y is child of x
      x.low_point = min{y.low_point, x.low_point}
      if y.low_point ≥ x.level:
        x.articulation = true;
    else: x.low_point = min{y.level,x.low_point};
```

10. **Theorem**: can find all articulation points and all bi-connected components in O(n+m) time.

11. **Bipartiteness DFS** $O(n+m)$

```
def TestBipartiteness(G):
  compute DFS-tree, compute level ∀u ∈ V
  if ∃{u,v}∈E with both u,v on odd/even level:
    return false
  else: return true
```

12. **Claim:** G is bipartite iff every edge in G is between a vertex on odd level and vertex on even level of DFS tree.

Efficient Tree & DAG Algorithms

1. Undirected graphs efficient if trees or forests, with top-down or bottom-up in $O(n)$ time. Directed graphs efficient if DAG, with topological sort + DP.

2. **Longest (simple) path** is NP-hard, but $O(n)$ for trees and $O(n+m)$ for DAGs.

3. **Longest simple path in forest**: run DFS from root r, then for each vertex $u \in V$: Let LPI(u) = longest path fully in subtree rooted at u, LPT(u)=longest path from u to any descendant. Then in $O(n)$ time return $LPI(r)$:
$$\mathrm{LPI}(u) = \max\begin{pmatrix} \max_{1\leq i\leq s} \mathrm{LPI}(v_i), \\ 2 + \max_{1\leq i<j\leq s} (\mathrm{LPT}(v_i) + \mathrm{LPT}(v_j)) \end{pmatrix}$$
$$\mathrm{LPT}(u) = 1 + \max_{1\leq i\leq s} \mathrm{LPT}(v_i)$$

4. **Longest simple path in DAG** takes $O(n+m)$ time:

Longest Simple Path in DAG $O(n+m)$

```
1. Topologically sort G
2. for i in range(n, 1): LP(i) = 0;
     for each edge (i,j)∈ E: # edge relaxation
       if 1+LP(j)>LP(i): LP(i)=1+LP(j)
3. return largest LP(i)
```

Aspect	Forest (tree)	DAG
Traversal	single DFS+DP	topo-sort,edge relax.
State per node	LPI,LPT	LP
Time	$O(n)$	$O(n+m)$

Graph Colouring

1. **Proper vertex-colouring** of G with set of colours C is a function $c: V \to C$ s.t. $c(u) \neq c(v)$ for all $(u,v) \in E$. If $|C| = k$, then called (proper) **k-colouring**.

2. **Chromatic number** $\mathcal{X}(G)$ of graph G is the smallest k s.t. there exists a k-colouring of G.

3. **Lemmas**: $\mathcal{X}(G)$=2 if G bipartite; $\mathcal{X}(T)$=2 if T is tree; $\mathcal{X}(C)$=3 for odd cycle C and $\mathcal{X}(K_n)$=n complete graph

4. **Lemma** : If graph G has subgraph G': $\mathcal{X}(G) \geq \mathcal{X}(G')$

5. **Clique-number** $\omega(G)$ is cardinality of largest subset $K \subseteq V$ s.t. $G[K]$ is a complete graph, where $G[K]$ is subgraph of G induced by vertex set K.

6. **Lemma**: if graph contains k-clique K_k: $\mathcal{X}(G) \geq \omega(G)$.

7. **Lemma**: GreedyColours colours G with $\leq \Delta(G)+1$ colours, where $\Delta(G)$ denotes maximum degree of G.

Greedy Colouring

```
def GreedyColouring(G): initialise i = 1;
  while i≤ n: # c(n) is colour number of node n
    c(v_i) = min_{j<i}{ℕ \ {c(v_j) : {v_i, v_j} ∈ E}}
    i += 1
```

8. **Theorem**: $\mathcal{X}(G) \leq \Delta(G) + 1$ for any graph G.

Planar Graphs

1. Graph G is **planar** if it can be drawn in plane $\mathbb{R}^2$ with every vertex v drawn as a point $f(v) \in \mathbb{R}^2$ and every (u, v) edge as continuous curve between $f(u)$ and $f(v)$ s.t. no two edges intersect except possibly, end points.

2. **Plane graph** or **planar embedding** is a planar drawing of a planar graph. **Faces** of this drawing are connected components of $\mathbb{R}^2$ after we delete drawing's vertices, edges. ***Note***: *square has 2 faces: inside, outside.*

3. **Euler's Formula**: Let $G = (V, E)$ be a connected planar graph. Let F be the set of faces of G's planar embedding, and cc_G is # connected components in G:

$$|V| + |F| = \begin{cases} 2 + |E| & \text{if } G \text{ connected} \\ 1 + cc_G + |E| & \text{if } G \text{ not connected} \end{cases}$$

Proof by induction: **Base case**: if G is acyclic, then $|F|$=1, so theorem holds since G is tree and $|E| = |V|-1$. **Inductive step**: assume G has a cycle C, let $e \in C$, then delete e from G, resulting in G^*. Now G^* has $|V|$ vertices, $|E| - 1$ edges and $|F| - 1$ faces, so

$$|V| + (|F| - 1) = (|E| + 1) \Rightarrow |V| + |F| = 2 + |E| \quad \square$$

(*Can keep removing cycles until base case reached.*)

4. So, every tree, cycle and K_4 is planar, but K_5 or $K_{3,3}$ is not. Note: $|F|$ is independent of planar embedding.

5. **Theorem**: for any simple connected planar graph G with $n > 2$ it holds that $|E| \leq 3|V| - 6$.

Proof: every face has ≥ 3 edges bounding it. Every edge bounds ≤ 2 faces, so $2|E| \geq 3|F|$, euler's formula: $|E| = |V| + |F| - 2 \Rightarrow 3|E| = 3|V| + 3|F| - 6 \leq 3|V| + 2|E| - 6$

6. **Corollary**: K_5 is not planar.
Proof: K_5 has 5 vertices, each vertex has 4 neighbours, so $n = 5, m = 10$, must hold: $m \leq 3n - 6$, but $10 \geq 3 \cdot 5 - 6$, contradiction. $\square$

7. **Lemma**: every simple planar graph has vertex of degree at most 5. **Proof** by contradiction: FTSOC suppose $\forall v \in V : d(v) > 5$, then $2|E| = \sum_{u \in V} d(u) \geq \sum_{u \in V} 6 = 6|V|$ but $\forall n > 2$: $|E| \leq 3|V| - 6$, contradiction. $\square$

8. **Corollary**: Every simple planar graph is ***6-colourable***.
Proof: **Base case**: trivially holds for $n \leq 6$. In $O(n)$: **inductive step**: find vertex v with $d(v) \leq 5$. Recursively colour $G - v$ using 6 colours. Now return v and choose out of $6 - d(v) = 1$ remaining colours, assign to v as graph remains simple planar upon vertex deletion. $\square$

9. **Lemma**: Every simple planar graph is ***5-colourable***.
Proof by induction: **Base case:** if G has vertex v of degree $d(v) \leq 4$, do induction same as 5-colourability.
Otherwise $\exists v$ s.t. $d(v)$=5, remove v from G and colour obtained graph, bring v back. If among 5 neighbours, not all 5 colours are used - assign missing colour to v.
Otherwise wlog v has 5 neighbours $u_1, ..u_5$ meaning vertex u_i has colour i. Try to replace u_1's colour with colour 3. Consider subgraph H of G induced by vertices with colours $1, 3$. If u_1 is disconected from u_3 in H, then consider component of H containing u_1, swap colours $1, 3$ in that component and colour v with colour 3.
Otherwise take $u_1 \rightsquigarrow u_3$ path π in H. If add edges $\{v, u_1\}, \{v, u_3\}$ to π then obtain **Jordan curve** separating u_2 from u_4. So graph induced by vertices coloured 2, 4 can't have u_2 and u_4 connected, so in that graph's component containing u_4 swap colours 2,4. Colour v with colour 4, as neighbours only have 4 colours. $\square$

10. **Theorem**: Every simple planar graph is ***4-colourable***.

11. **Dual graph** $G^* = (V^*, E^*)$ of planar graph $G = (V, E)$: each vertex $u \in V^*$ corresponds to a face in G. Two vertices in G^* are connected by an edge if corresponding faces in G have boundary edge in common. G^* is planar, $|V^*| = |F|, |E^*| = |E|, |F^* = |V|$

12. **Theorem**: any map in a plane can be coloured using four colours s.t. regions sharing a common boundary (other than single point) do not share same colour.
Proof: consider **duals** of planar graphs.

13. **Kuratowski's Theorem**: A graph G is planar iff it has no minor isomorphic to K_5 or $K_{3,3}$- if it's impossible to subdivide edges of either and add edges and vertices to obtain G.

MST

1. Graph S=(V_S, E_S) is **spanning subgraph** of G=(V, E) iff it covers all its vertices $V_S = V$ and uses some of its edges $E_S \subseteq E$.

2. **Spanning tree (ST)** is a maximal (adding any new edge would create a cycle) connected spanning subgraph of G with no cycles and $|\mathbf{V}| - \mathbf{1}$ **edges**.

3. **Minimum Spanning Tree (MST)** is ST of undirected connected graph G with weighted edges that has the **minimum weight**.

4. **Meta-Algorithm**: build ST edge by edge by including a blue edge or excluding a red one until the tree is built.

Blue Rule: partition V=$V_1 \cup V_2$, subgraphs $G_1, G_2 \subseteq G$ vertex-induced by V_1, V_2 connected by **uncoloured** edges, choose **lightest** such edge, colour it blue.

Red Rule: select any simple cycle containing no red edges, colour the maximum weight uncoloured edge red.

MST Meta Algorithm

```
def Meta-Algorithm(G):
    Initialise: all edges ∀e ∈ E are uncoloured
    while there are uncoloured edges:
        apply either blue or red rule
    return tree formed by blue edges
```

5. **Colouring invariant**: there exists a MST that contains all blue edges and no red edges.

6. **Theorem (preserving invariant)**: after meta algorithm colours all edges, the blue edges will form a MST. *In other words, the meta algorithm is correct.* **Proof**:

   Blue rule: Let T be MST (before colouring edge e). If e is blue, and is already in T, then the invariant holds. If it isn't then there must exist a path in T connecting e's endpoints, containing some edge e^*. Since e^* is uncoloured and weight $w(e^*) \geq w(e)$, can replace e^* with e preserving the MST property. □

   Red rule: if e is red, then if $e \notin T$ then the invariant holds, otherwise deleting e in T creates subtrees V_1, V_2. Since e was part of cycle when coloured red, $\exists$ some uncoloured edge e^* in that cycle with $w(e^*) \leq w(e)$, so replacing e with e^* preserves MST property. □

   Termination: FTSOC, assume algorithm stops while some edges are still uncoloured, but initially blue edges form a forest. For any uncoloured edge e: if its endpoints are in **same** blue tree, red rule removes e, if in different blue trees, applying blue rule adds e, so uncoloured edge always allows a rule to be applied, contradiction. □

7. Any MST algorithm implementing meta algorithm framework is correct. A few such algorithms:
   - **Kruskal's:** $O(m \log n)$, or $O(m\alpha(n))$ for int weights.
   - **Prim's:** $O(n^2)$, or $O(m \log n)$ with simple priority queues, or $O(m + n \log n)$ with Fibonacci heaps.
   - **Borůvka's:** Suitable for parallel implementations.
   - **Round-robin:** $O(m \log \log n)$ or $O(n)$ in planar graphs.

MST Data Structures, Union-Find

1. **Union-Find** data structure maintains disjoint dynamic sets $S = \{s_1, ..s_k\}$ supporting following three operations:
   - `Make-Set`(x) creates new set whose only member is x.
   - `Union`(x, y) unites disjoint sets containing x, y into a new set that is the union of the two sets.
   - `Find`(x) returns **representative** of set containing x.

   Typically need $O(n)$ `Union` and $O(m)$ `Find` operations.

2. In union find, two vertices are in the same blue tree if their `Find` returns the same representative. Blue rule: merge two trees into one, red rule: do nothing. Each set s_i contains vertices from the same blue tree.

3. **Characteristic vector** is the simplest implementation of union-find, write $\mathcal{X}^{(i)}$ be the representative of set containing i. Basically array where index is vertex number and value is that vertexes representative.

   `Make-Set`(x): $\mathcal{X}^{(x)} = x$ `O(1)` **`Find`**(x): return $\mathcal{X}^{(x)}$ `O(1)`
   `Union`(x, y): for i=1 to n: $\mathcal{X}^{(i)}$=$\mathcal{X}^{(y)} \to \mathcal{X}^{(i)}$:=$\mathcal{X}^{(x)}$ `O(n)`

   Kruskal algorithm calls `Find` m times and `Union` $n-1$ times, so overall $O(m+n^2)$ using a characteristic vector.

4. **Tree structure** data structure is a better union-find implementation. Have collection of trees, with the root being their representative, each node has directed edge to their parent. Parent has a self-loop to itself. Don't need to represent trees, just function (relationship) `parent`.

Tree data structure for Union-Find

```
def Make-Set(x):
    create new tree rooted at x, parent(x := x)
def Union(x,y):
    parent(Find(x)) := Find(y)
def Find(x):
    y := x;
    while y ≠ parent(y) { y := parent(y) }
    return y
```

inefficient: n calls to `Make-set` and `Union`, m calls to `Find`, overall $O(n(n + m))$. Maintain **balanced height** trees to ensure $O(\log n)$ height, use path compression to keep trees shallow, as time of `Find` = tree height.

5. **Path compression**: each time `Find`(x) is performed, change `parent` link for all nodes on the path from x to the root, to point to the root. So (*start*) $x \rightsquigarrow a \rightsquigarrow b \rightsquigarrow p$ (*parent*) becomes $x \rightsquigarrow p, y \rightsquigarrow z$.

6. **Weight/Height/Rank union rule**: in `Union`(x, y) let:
   1) ***Weight:*** # nodes in the tree containing x be $\geq$ # nodes in the tree containing y;
   2) ***Height:*** height of tree containing x be $\geq$ height of tree containing y;
   3) ***Rank:*** rank (height not updated by path compression) of tree containing x be $\geq$ rank of tree containing y;
   Set `parent(Find(`y`)) = Find(`x`)`. Union now takes $O(1)$.

7. **Inverse Ackermann's function** is very small, $\alpha(n) \leq 4$ for $n \leq A_4(1)$. Defined as: $\alpha(n)$=min$\{k : A_k(1) \geq n\}$
   Ackermann's function $A_k(j) = \begin{cases} j+1, & k=0 \\ A_{k-1}^{j+1}(j), & k>0 \end{cases}$

8. **Theorem**: sequence of n `Make-Set` and m `Find` and `Union` operations performed with ***path compression*** rule and either ***weight or rank union*** rule takes $O(n + m \cdot \alpha(n))$ time - **pseudopolynomial**.

MST Algorithms

1. **Kruskal's Algorithm** runs in $O(n+m \log n)$ in general, and $O(m \cdot \alpha(n))$ for integer weights $\{1, 2, ..n^2\}$. Assume input graph is connected $m \geq n - 1$ & simple $m \leq \binom{n}{2}$.

Kruskal Algorithm $O(m \log n)$ **or** $O(m \cdot \alpha(n))$

```
def Kruskal(G):
    Initialization: all edges are uncoloured
    sort all edges in non-decreasing order
    for all edges in the non-decreasing order:
        # apply either blue or red rule;
        if Find(u) ≠ Find(v):
          colour(u,v) blue, Union(u,v)
        else colour (u,v) red
    return the tree formed by blue edges
```

2. **Prim's Algorithm** easily done in $O(n(n+m))$, make faster: for any vertex $v \in V \setminus T$, define:

$$d(v) = \min\{w(v,u) : u \in T, (v,u) \in E\}$$
$$\pi(v) = u \text{ s.t. } w(v,u) = d(v), u \in T, (v,u) \in E$$

where $d(v)$ is cost of the lightest edge between v and MST T, and $\pi(v)$ is the endpoint of that edge.

Initialisation: T has only vertex s, so $\forall v \in V \setminus \{s\}$:

- If $\{s,v\} \in E$, set $d(v) = w(v,s)$ and $\pi(v) = s$
- Else, set $d(v) = \infty$ and $\pi(v) = \text{NIL}$

Later: find $v \in V \setminus T$ minimising $d(v)$. If v joins T, then perform Decrease-Key(v)

Prim's Algorithm $O(n \log n + m)$

```
def Prim(G):
    Initialization: all edges are uncoloured
    repeat n - 1 times:
        Let T be a blue tree containing s;
        Select a min-weight edge e incident to T;
        Colour e blue;
    return the tree formed by blue edges
```

taking $O(n^2)$ time with an array, $O(n + m \log n)$ using PQ/heap, and $O(n \log n + m)$ with Fibonacci heap:

3. **Boruvka Algorithm** is good for good parallelism.

Boruvka MST Algorithm

```
def Boruvka(G):
  Initialization: all edges are uncoloured
  repeat until a single tree contains all nodes:
    for every blue tree T:
      Select a min-weight edge e incident to T;
      Colour e blue;
  return the tree formed by blue edges
```

4. **Round-Robin Algorithm** is good for planar graphs.

Round-Robin MST Algorithm $O(m \log n)$

```
def Round-Robin(G):
  Initialization: Q := V # queue of V's partition
  repeat n-1 times:
    let A be first element of Q
    apply blue rule to A and V \ A
    let {x,y} be the new blue edge
    let x ∈ A and y ∈ B for some set B in Q
    delete A and B from Q
    add A ∪ B to end of Q
return tree formed by blue edges
```

Stage 1: ends when the last element originally in Q is removed.
Stage *i*: ends when all elements that were in Q at the start of the stage have been removed.

5. **Lemma**: sets entering stage k have size $\geq 2^{k-1}$; produced in that stage have size $\geq 2^k$.
Corollary: There are at most $\log n$ stages.
Claim: Each stage can be implemented in $O(m)$ time.

6. **Modified Round-Robin (RR)**: simple graphs have $\leq 3n$ edges, after each stage, contract all blue trees into supervertices, delete edges between two vertices in same tree, and all but the lightest edges between supervertices.

7. **Claim**: contraction of simple planar graphs gives a simple planar graph.

8. **Round-Robin Algorithm for planar graphs**

Round-Robin for Planar Graphs $O(n)$

```
def Round-Robin(G):
  k := 1; G_k := G
  while G_k has more than one vertex:
    run a single stage of Round-Robin
    contract G_k into G_{k+1}
    k := k + 1
```

At each stage, we consider graph $G_k = (V_k, E_k)$ with $|V_k| \leq n/2^{k-1}$ vertices, so total runtime is:

$$\sum_{k=1}^{\log n} O(|V_k|) = \sum_{k=1}^{\log n} O(n/2^k) = O(n)$$

9. **Theorem**: RR algorithm for planar graphs can be implemented in $O(n)$ time.

10. **Fast Round-Robin MST Algorithm**: for supergraph with $\leq 2m/\log n$ number of groups:

Run Round-Robin $O(\log\log n)$ times (so stop early) so that the original vertices will be connected into $O(n \div \log n)$ groups (supervertices). Divide edges incident to each such supervertex into groups of size $\leq \log n$ and sort them by weight. Run round-Robin again, but this time inspect the edges in groups (trees) - need find the cheapest edge leaving the tree (already sorted, so cheap), but visiting each neighbour to check if they're within the set is too expensive - sort by value first, and only then check if it's leading outside the tree:

$$\sum_{i=1}^{k} \frac{\lceil \deg(U_i) \rceil}{\log n} \leq \sum_{i=1}^{k} \frac{\deg(U_i)}{\log n} + 1 = \frac{m}{\log n} + 1 \leq \frac{m}{\log n} + \frac{n}{\log n} = \frac{m+n}{\log n} \leq \frac{2m}{\log n}$$

there's also some near-constant processing time per each group, approximated at $O(\log\log n)$. For sparse graph $m \sim n$ Round Robin isn't ideal.

Matching

1. **Matching problem**: for undirected graph $G = (V, E)$, **matching** H is subset of edges s.t. no two edges in H share an end-point. **Max matching**:
Goal 1: find a maximum cardinality matching.
Goal 2: if edges are weighted ($\forall e \in E$ weighs $w_e \geq 0$), find maximum weight matching (summed edge weights).

2. **Bipartite Matching**: for undirected bipartite graph $G = (L \cup R, E)$ (left, right sets), $M \subseteq E$ is matching if each node appears in ≤ 1 edge in M. **Perfect matching** if $|M| = |L| = |R|$.

3. **Max-Flow problem**: given undir $G = (V, E)$ with special source s and target/sink t vertices, capacity on every edge $c : E \to \mathbb{R}_{>0}$, find $s - t$ flow f of maximum value.

$$\forall e \in E : 0 \leq f(e) \leq c(e)$$

$$\forall v \in V \setminus \{s, t\} : \sum_{(u,v)\in E} f(u, v) = \sum_{(v,w)\in E} f(v, w)$$

$$\text{Value of } f = \sum_{(s,u)\in E} f(s, u) = \sum_{(v,t)\in E} f(v, t)$$

maximum flow can be found in polynomial time $O(nm)$.

4. **Theorem**: size of a maximum matching in G is at most, and at least (so equal) the value of a max flow in G' where $G' = (L \cup R \cup \{s, t\}, E' \cup \{L \times s, R \times t\})$ (edge from source to $\forall l \in L$ and from $\forall r \in R$ to sink t).

5. **Integrality theorem**: if k is an integer, then can assume the flow f is either 0 or 1.

6. **Free vertex** with respect to M is one not incident to any edge of a matching M in graph G.

7. **Alternating path** is a path P in G s.t. edges in M alternate with edges not in M.

8. **Augmenting path**: alternating acyclic path for matching M that starts and ends at distinct free vertices.

9. **Berge Theorem**: A matching M is a maximum matching iff there is no augmenting path with respect to M.
Proof: ($\Rightarrow$) if M is maximum then there is no augmenting path w.r.t M. Can switch matching and non-matchingedges along the pat, giving matching $M' = M \oplus P$ (XOR) with larger cardinality.

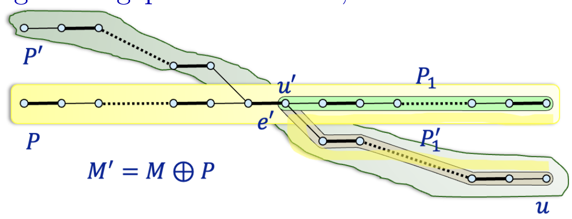
($\Leftarrow$) Suppose there is a matching M^* with $|M^*| > |M|$. Consider graph H with edge set $M^* \oplus M$. Each vertex in H can be incident to ≤ 2 edges (one from M, one from M^*). So, CCs of H are alternating cycles/paths. More of edges are from M^* since $|M^*| > |M|$, so there's only 1 CC that's a path P for which both endpoints are incident to edges from M^*, but then P is alternating path w.r.t. M. □

Maximum Bipartite Matching

1. **Alternating Tree construction**: partition into odd and even vertices using BFS. Take vertex y, now cases:
1) y is free vertex not contained in T, then found an augmenting path.
2) y is a matched vertex, neither y nor `mate`(y) are in T, grow the tree by adding such matched edge.
3) $y \in T$ as an odd vertex, ignore successor y.
4) $y \in T$ as an even vertex, can't ignore, odd length cycle, which is not possible in bipartite graphs.

2. **Matching algorithm**: as long as you find augmenting path, use it to augment the matching, else is maximum. Constructing alternating tree takes $O(n+m)$. Since each matching augmentation increases the size of matching, and any matching has size $\leq \lfloor n/2 \rfloor$, will need $\leq \lfloor n/2 \rfloor$ augmentations, giving overall $O(n^2(n + m))$ runtime.

3. **Theorem**: Let M be a matching in graph G, and u be a free vertex w.r.t M. Now, let P be an augmenting path w.r.t M and $M' = M \oplus P$ be matching resulting from augmenting M with P. If there was no augmenting path starting at u in M, then there's no augmenting path starting at u in M'.
Proof by contradiction: FTSOC assume there is an augmenting path P' w.r.t M' starting at u.
If P' and P are node-disjoint, P' is also an augmenting path w.r.t M, contradiction. Otherwise, let u' be the first node on P' that is also on P. Now $P'_1 \circ P_1$ is an augmenting path w.r.t. M, contradiction. □



4. **Naïve Bipartite Matching**: Start with an empty matching and mark all vertices in L as free. For each free $r \in L$, ***grow an alternating tree*** until you either reach a free $y \in R$ (then augment along the discovered path and decrement the free count) or exhaust all possibilities. Repeat until no augmenting path exists.

Naïve bipartite max-matching $O(n(n + m))$

```
def BipartiteMatch(G): # G=(L ∪ R,E)
  for v in V: mate[v] = 0 # empty matching
  free = |L| # initialise to be unmatched
  # try each root until all matched
  for r in range(1, |L|) while free > 0:
    if mate[r] == 0: # if unmatched
      parent[v] = None for all v # cleanup
      Q = [r]; aug = False; # enqueue root
      # grow alternating tree till augment found
      while Q ≠ [] and not aug: # or is max-matched
        x = Q.pop(0)
        for y in neighbours(x):
          if mate[y] == 0: # unmatched
            augment(parent, y) free -= 1 aug = True
            break # match and restart
          elif parent[mate[y]] is None:
            parent[y] = x # already matched
            Q.append(mate[y]) # keep growing
```

5. **Hopcroft-Karp fast max-matching** can find maximum matching in bipartite graph in $O(\sqrt{n}(n + m))$.

6. Ensure that path lengths all grow in each phase, so construct maximal set Π of disjoint augmenting paths w.r.t M. Denote $M \oplus \Pi = M \oplus (\oplus_{P \in \Pi} P)$.

7. **Hopcroft-Karp fast max-matching algorithm**:
   1. Initialise $M = \varnothing$
   2. Repeat until no augmenting paths exist:
      - Build alternating tree rooted at unmatched vertices in L with BFS, basically a Trie (don't re-add nodes
      - For each unmatched node r_i in L: Run DFS rooted at r_i always moving down the BFS tree: using (u, v) edges with $d(v) = d(u)+1$, to find shortest augmenting path to the first unmatched vertex in R. Then XOR (augment) the path and remove its vertices to ensure vertex-disjoint paths.
   3. Return M.

   Runs in $O(\sqrt{n}(n + m))$, since each phase increases length of shortest augmenting path by 1, so after $\sqrt{n}$p hases by lemma will only have $\sqrt{n}$ paths left, so overall augment $\leq 2\sqrt{n}$ times, each augmentation taking $O(n + m)$ from BFS/DFS. □

8. **Lemma**: Let M^* be a maximum matching and let M be any matching in G. If length of the shortest augmenting path w.r.t. M is k, then $|M^*| - |M| \leq \frac{|V|}{k}$.
   Proof: consider graph $G^* = (V, M \oplus M^*)$. It contains $\geq |M^*| - |M|$ augmenting paths with respect to M, each of length $\geq k$. Total length of these paths is $|V|$, so there are $\leq \frac{|V|}{k}$ of them. □

9. **Lemma**: Let k be length of shortest augmenting path w.r.t M. Let Π be maximal set of shortest disjoint augmenting paths w.r.t. M (all of length k), then the shortest such path length w.r.t. $M \oplus \Pi$ is $> k$.
   Proof: Consider shortest path P w.r.t $M \oplus \Pi$. If P doesn't intersect any path from Π then its length is $> k$ as Π is maximal. Else suppose P intersects paths $P_1, ..P_t$ from Π. Combine these paths to construct $t+1$ new augmenting paths R_i w.r.t M, now $|R_j| \geq k$. Total length of $t + 1$ paths is shorter than total length of paths P_i:
   $$\sum_{i=1}^{t+1} |R_i| < |P| + \sum_{i=1}^{t} |P_i| = |P| + t \cdot k$$
   This and $|R_j| \geq k$ implies that $|P| > k$. □

10. For bipartite graph $G = (L \cup R, E)$ and matching M, define $G_M(L \cup R, E_M)$ as:
    $E_M = \{(u, v) : \{u, v\} \in E \setminus M, u \in L, v \in R\} \cup \{(u, v) : \{v, u\} \in M, v \in L, u \in R\}$
    Use this to find layered graph G^*_M constructed out of G_M. Let L^* be free vertices in L, and $d : V \mapsto N$ be distance $d(v)$ from v to vertices in L^*. Then graph $G^*_M = (L \cup R, E^*_M)$ contains the following edges:
    $$E^*_M = \{(u, v) : (u, v) \in E_M \text{ and } d(u) + 1 = d(v)\}$$

11. **Lemma**: every path in G^*_M that starts in L^* is a shortest path in G_M.

12. **Perfect matching** covers all vertices from L.

13. **Hall's Theorem**: A bipartite graph $G = (L \cup R, E)$ has a perfect matching iff for all sets $S \subseteq L : |\Gamma(S)| \geq |S|$.
    Here, $\Gamma(S)$ is the set of vertices in R that have a neighbour in S. *I.H:* $\forall S \subseteq L \wedge |\Gamma(S)| \geq |S| \rightarrow G$ *perf matched.*
    Proof: ($\Rightarrow$) this condition is necessary, as otherwise not all nodes would match.
    ($\Leftarrow$) For $|L| > 1$: pick arbitrary $v \in L$ that has at least 1 neighbour $u \in R$. Match and remove v, u and their incident edges. Now find matching of size $|L| - 1$ in smaller graph induced on $L \setminus \{v\}, R \setminus \{u\}$ by induction.
    Won't work if $\exists Q \subseteq L \setminus \{v\}$ has $< |Q|$ neighbours in $R \setminus \{u\}$, take such smallest set Q. Then $|\Gamma(Q)| = |Q|$, so $\forall U \subseteq Q : |\Gamma(U)| \geq |U|$ for the I.H. to hold.
    By induction, $\exists$ perfect matching between $Q, \Gamma(Q)$. For $\forall A \subseteq L \setminus Q$, by I.H., $A \cup Q$ has $\geq |A| + |Q| = |A| + |\Gamma(G)|$ neighbours in R. So A has $\geq |A|$ neighbours in $R \setminus \Gamma(Q)$, so $\exists$ perfect matching between $L \setminus Q$ and $R \setminus \Gamma(G)$. □

14. **k-Regular** bipartite graph: $\forall v \in V : \deg(v) = k$.

15. **Lemma**: For every $d \geq 1$, every d-regular bipartite graph has a perfect matching.
    Proof: take any vertex set $S \subseteq L$. Since G is d-regular, cumulative degree $\sum_{u \in S} \deg(u) = d \cdot |S|$, so there are $\Gamma(S) \geq d \cdot |S|$ neighbours, but # edges incident to $\Gamma(S)$ is $\sum_{u \in \Gamma(S)} \mathbf{deg}(u) = d \cdot |\Gamma(S)|$, yielding $|\Gamma(S)| \geq |S|$. □

16. **Lemma**: For every $d \geq 1$, every $d - regular$ bipartite graph has exactly d edge-disjoint perfect matchings.
    Proof: by Hall's theorem, every d-regular bipartite has a perfect matching. Remove one such matching and obtain $(d - 1)$-regular bipartite graph. Repeat to recursively find d such edge-disjoint perfect matchings. □

Weighted Bipartite Matching, VC

1. **Vertex Cover (VC)** is set C of vertices s.t. all edges $e \in E$ are incident to at least one vertex of C. No edge is completely contained in $V \setminus C$ (outside of cover).

2. **Weak duality**: any vertex cover is at least as large as the maximum size matching.

3. **König's Theorem**: for any bipartite graph, maximum size of a matching is equal to the minimum size of a VC.
   Proof: fix a maximum matching M. Let Q_L be all vertices reachable in G_M from free vertices in L. Then $C^* = (L \setminus Q_L) \cup (R \cap Q_L)$ is a VC with $|C^*| = |M|$.
   All vertices in L are in $L \cap Q_L$. All free vertices in R are in $R \setminus Q_L$ as otherwise would get augmented path w.r.t M contradicting M's maximality.
   There's no edge from M between vetex $x \in L \setminus Q_L$ and $y \in R \cap Q_L$, as otherwise x would be in Q_L (matched). So, every vertex in C^* is matched in M and corresponding matching edges are distinct, so $|C^*| \leq |M|$. □

4. **Weighted Bipartite matching**: undirected bipartite graph $G=(L \cup R, E)$, each $e=(l,r) \in E$ has ***edge weight*** $w(e) \geq 0$.

5. **Assignment problem**: find a matching of maximum weight (sum of weights). Wlog assume $|L| = |R| = n$ and $\exists$ edge between every pair of nodes $(l,r) \in L \times R$.

6. **Tight subgraph** $H(\overrightarrow{x})$ of G only contains edges that are **tight** $(x_u + x_v = w(e))$ w.r.t the node weighting $\overrightarrow{x}$.

7. **Node-weighing** $\overrightarrow{x}$: each $v \in V$, has weight $x_v \geq 0$. **Invariant**: node weights dominate edge weights as:
$$x_u + x_v \geq w(e) \text{ for every edge } e = (u,v).$$
Try to compute perfect matching in $H(\overrightarrow{x})$. Matching weight is total node weight $X = \sum_{v \in V} x_v$, optimal if:
$$\sum_{(u,v)\in M} w(u,v) \leq \sum_{(u,v)\in M} (x_u + x_v) \leq X = \sum_{v \in V} x_v$$

8. **Naïve re-weight:** to reduce total node weight $X = \sum_v x_v$ while maintaining the invariant $x_u + x_v \geq w(u,v)$ for all $(u,v) \in E$, and ideally increase size of the maximum matching in $H(\vec{x})$.

   Let $S \subseteq L$ be a subset violating Hall's condition: $|\Gamma(S)| < |S|$ where $\Gamma(S)$ are the neighbours of S in $H(\vec{x})$.

   - Increase node weights by δ for each $v \in \Gamma(S)$ and decrease by δ for each $u \in S$.
   - This makes edges from S to $R \setminus \Gamma(S)$ strictly lighter (may become tight later), without violating invariant.
   - All tight edges remain tight; none become invalid.
   - Eventually, new edges may become tight and be added to $H(\vec{x})$, increasing its connectivity and potential matching size.

   After changing weights, there's at least 1 more edge edge leaving $L \cap Q_L$, after $\leq n$ re-weights can do augmentation. Re-weighting takes $O(n^2)$, augmentation $O(n)$, so overall $O(n^4)$. Can do better.

9. **Theorem**: finding maximum wight matching in bipartite graphs can be done in $O(n^3)$ time.

10. **Hungarian Algorithm**: find a minimum-weight perfect matching taking $O(n^3)$ time.

    1) Reduce to max-weight perfect matching: original cost of matching M is $W(M) = \sum_{e \in M} w(e)$. If M^* is any other perfect matching, then $\sum_{e \in M^*} w(e) = W(M^*) \geq W(M) = \sum_{e \in M} w(e)$. Now swap edge weights from w to $-w$, to get $\hat{W}(M^*) = -W(M^*) \leq -W(M) = \hat{W}(M)$.

    2) Ensure all weights are non-negative. Swap weights from $w(e)$ to $w(e) = \min_{e'} w(e')$ where:
    $$(1)\ \hat{w}(e) = w(e) = \min_{e' \in E} w(e') \geq 0$$
    $$(2)\ \hat{W}(M^*) = W(M^*) - |M^*| \cdot \min_{e'} w(e') \leq W(M) - |M| \cdot \min_{e'} w(e') = \hat{W}(M)$$
    Add additional vertices to make both sides of the same size. Add new edges to make G complete bipartite.

Max-matching in general graphs

1. Construct alternating tree T, now since $G = (V,E)$ is not necessarily bipartite, can no longer ignore the case where x, y are on the same even layer, and $(x,y) \in E$. Let w be their least common ancestor (LCA).

2. **Blossom** is the odd cycle induced by alternating tree paths $w \rightsquigarrow x, w \rightsquigarrow y$ and edge (x,y). Here, even vertex w is the **base** of the blossom.

3. **Shrinking blossoms**: If during alternating tree construction we discover a blossom B, replace G with $G' = G/B$ obtained by replacing all vertices in B with a supervertex b (***shrink***). Then, find an augmenting path through that contracted node in G', augment path, and replace supervertex b with original vertices in B (***lift***).

   - **Initialize.** Set all vertices free, `mate[v]` $= 0$.
   - **For each free root** r**.** Start BFS on an alternating tree: label r even ($d[r] = 0$), enqueue.
   - **Explore** x**.** For each unexamined neighbour y:
     (A) y even in tree: *blossom* found $\to$ contract the odd-cycle into b, update labels/distances.
     (B) y odd and matched: grow tree via $y \to z = \text{mate}[y]$, label z even, enqueue.
     (C) y odd and free: augmenting path found, **stop BFS**.
   - **Augment & expand.** Reconstruct via parent pointers, flip matches, expand any contracted blossoms in reverse.
   - **Repeat.** Continue with next free r until no augmenting path exists.

   Runs in $O(n^2(n+m))$, *can* $O(n(n+m))$ *or* $O(\sqrt{n}(n+m))$

4. Tree edges of T connected to node u in B become tree edges in T' connecting u to b. Matching edge (***stem***) connecting node u not in B to a node in B becomes matching edge in M'. Nodes connected in G to at least one node in B become connected to b in G'.

5. **Lemma 1**: matching M in G induces matching M' in G':
$$M' := \{\{u,v\} : \{u,v\} \in M \wedge u,v \in V \setminus B\}$$
$$\cup \{\{u,b\} : \{u,x\} \in M \wedge u \in V \setminus B \wedge x \in B\}$$
   Every node in G' has at most 1 incident matching edge. All nodes in $V \setminus B$ have same # matching edges incident in M in G, so matching condition fulfilled in them. For node b, can only have at most 1 incident matching edge since only node in blossom that can have incident matching edge is the base w. □

6. **Lemma 2**: Current alternating tree T w.r.t. matching M induces alternating tree T' in G' w.r.t matching M':

   For every edge $\{u,v\} \in T$ and $u,v \in V \setminus B$, have corresponding $\{u,v\} \in T'$; for every such edge with $u \in V \setminus B$ and $v \in B$, introduce edge $\{u,b\} \in T'$. Node b becomes even node in T'; if root r of T is contained in B then b becomes root of T'.

Proof: Must show (i) T' is acyclic, (ii) connected, and (iii) every root–leaf path alternates.

(i) Acyclicity. If T' contained a cycle, expanding the contracted node b back to the blossom B would yield a cycle in T, contradicting its tree property.
(ii) Connectivity. Contracting the connected subgraph B to a single node b cannot disconnect T, so T' remains connected.
(iii) Alternation. On any root–leaf path in T':

- Edges whose endpoints lie both outside b correspond exactly to those in T and thus alternate.
- If the path uses an edge (u, b) and then (b, v), by construction $(u, b) \in M'$ and every other edge incident to b is not in M', so these two also alternate.

Hence T' is an alternating tree in G' w.r.t. M'. $\square$

7. **Lemma 3**: If G contains augmenting path w.r.t M starting at root r, then G' contains an augmenting path w.r.t M' starting at r (or b if $r \in B$).

   Proof: Let P be $r \rightsquigarrow s$ augmenting path in G w.r.t M.

   Case 1. $P \cap B = \emptyset$. Then P also lies entirely in G' and remains augmenting w.r.t. M'.

   Case 2. $P \cap B \neq \emptyset$. Since every vertex of $B \setminus \{w\}$ is matched, $s \notin B$. Let x be the last vertex of P in B, and let y be its successor on P. The subpath P_{xs} alternates (starting with a non-matching edge) and ends at free s, so in G' it induces an augmenting path from b to s.

   If $r \in B$, we are done. Otherwise, let S be the alternating "stem" in T from r to the base w. Flipping M along S gives a matching M^* of the same size and an augmenting path ending at $w \in B$, which reduces to the previous subcase. Prepending S then yields an augmenting path in G' starting at r. $\square$

8. **Lemma 4**: If G' contains augmenting path w.r.t M' starting at r (or b if $r \in B$), then there exists augmenting path in G w.r.t M starting at r.

   Proof: Let Q' be $u \rightsquigarrow s$ augmenting path in G' w.r.t M' where $u = r$ (or $u = b$ if $r \in B$).

   Case 1. $Q' \cap \{b\} = \emptyset$. Then Q' lies entirely in G and is augmenting w.r.t. M.

   Case 2. Suppose $b \in Q'$ but $r \notin B$. Then Q' traverses the two edges $(g, b) \in M'$, $(b, h) \notin M'$ so for some $b_q \in B$, it holds that $\{g, h\} = \{mate(w), b_q\}$. Since B contains an alternating path from w to b_q, splicing that path in place of the vertex b in Q' produces an augmenting path in G w.r.t. M.

   Case 3. $r \in B$. Then $u = b$, and the same "splice-in-the-stem" argument connects r via an alternating path in B to the neighbour q of b on Q', producing an augmenting path in G from r to s. $\square$

Max-matching in Planar graphs

1. **Planar Separator Theorem**: let $G = (V, E)$ be simple planar graph. One can partition V into $A, B, S \subseteq V$:
$$|S| \leq 2\sqrt{2n} \qquad \frac{n}{3} \leq |A|, |B| \leq \frac{2}{3}n$$
s.t. there is no edge between A, B in G. Can structure nodes in a $\sqrt{n} \times \sqrt{n}$ grid. Take $S = O(\sqrt{n})$, as the constants are unimportant, have impact only on runtime. Partition can be found in $O(n)$ time.

2. Since A, B, S are small, solve recursively: find max-matching in subgraphs of G induced by A, then by B, and find augmenting paths through S.
$$T(n) = T(|A|) + T(|B|) + |S| \cdot O(n)$$
$$T(n) \leq \max_{\frac{n}{3} \leq k \leq \frac{2n}{3}} \{T(k) + T(n-k) + O(n^{1.5})\}$$
$$T(n) \leq 2T(2n/3) + O(n^{\frac{3}{2}}) = \Theta(n^{\log_{3/2} 2}) \simeq \Theta(n^{1.71})$$
By induction: for constant c, have $T(n) \leq c \cdot n^{1.5}$. For any $\frac{n}{3} \leq k \leq \frac{2n}{3}$ have, differentiate to prove:
$$k^{1.5} + (n-k)^{1.5} \leq \left(\frac{n}{3}\right)^{1.5} + \left(\frac{2n}{3}\right)^{1.5} = \left(\frac{1}{3^{1.5}} + \frac{2^{1.5}}{3^{1.5}}\right) n^{1.5}$$
$$T(n) \leq c \cdot (k^{1.5} + (n-k)^{1.5}) + \alpha \cdot n^{1.5} \leq c \cdot n^{1.5} \cdot \frac{1 + 2^{1.5}}{3^{1.5}} + \alpha \cdot n^{1.5}$$
$$\text{Choose } c \geq \frac{3^{1.5} \cdot \alpha}{3^{1.5} - 1 - 2^{1.5}} \simeq 3.8 \cdot \alpha. \quad \square$$

3. **Weighted Planar Separator Theorem**: add weight function $w : V \to R_{\geq 0}$. Any subset $U \subseteq V$ weighs $W(U) = \sum_{u \in U} w(u)$. Total weight $W := W(V)$, now:
$$|S| \leq 4\sqrt{n} \qquad W(A), W(B) \leq \frac{2}{3}W$$

4. **Minimum Vertex Cover (Min-VC)**: find vertex cover of minimum cardinality. NP-hard, takes $2^{O(\sqrt{n})}$.

5. **Brute-force Min-VC algorithm**: for every subset $C \subseteq V$, check if C is a VC. Out of all sets C that form a VC, select the smallest one.

   Claim 1: the algorithm will find minimum-VC in G.
   Claim 2: runtime $\sum_{U \subseteq V} O(n+m) = 2^n \cdot O(n+m) = 2^{O(n)}$

   Proof: consider all 2^n subsets $C \subseteq V$, for each subset need $O(n+m)$ time to determine if C is a VC. $\square$

6. **Planar Separator Min-VC algorithm**:
   Outline: For each $U \subseteq S$, find min-VC C_U s.t. $C_U \cap S = U$, and choose overall min-VC over all such $U \subseteq S$.

   1) $C_U \cap S = U$ if $\forall$ edge e with both endpoints in S has ≥ 1 vertex from U, then there's ≥ 1 VC C of G s.t. $C \cap S = U$, e.g. $C = U \cup A \cup B$, find min such VC.

   2) Let $G\langle U \rangle$ be G with all edges incident to U removed. VC has to contain vertices $\forall u \in U$ and set Q of all $v \in A \cup B$ incident to at least 1 vertex in $S \setminus U$, so add Q, remove all edges incident on it.

   3) **Claim:** C is a vertex cover in $G\langle U \cup Q \rangle$ iff $C \cup U \cup Q$ is a VC in G. Isolates all $s \in S$, ignore them.

   Continue recursively.

7. For each subset $U \subseteq S$, have $T_U(n) \leq \Theta(n) + T(|A|) + T(|B|)$. Total running time is:
$$T(n) = \sum_{U \subseteq S} T_U(n) \leq 2^{|S|}(\Theta(n) + 2T(2n/3)) \leq 2^{O(\sqrt{n})} \cdot 2T(2n/3) \leq 2^{O(\sqrt{n})}.$$

Eulerian Paths & Hamiltonian Cycles

1. **Eulerian tour** visits each edge exactly once and returns to the starting vertex. G is **Eulerian** if it admits such tour. **Eulerian trail** need not return.

2. **Theorem**: Let G be ***undirected*** connected graph, may have parallel edges and self-loops. Then G has ***Eulerian tour*** iff degree $d(v)$ is even for every $v \in V$.

   Proof: ($\Rightarrow$) Let $in(v)$ be # edges incident to v used by C to enter v, and $out(v)$ - # to exit v. Now $in(v) = out(v)$ as tour must enter and exit through different edge each time. Hence, $d(v)$ must be even for every vertex $v \in V$.

   ($\Leftarrow$) Let G be connected graph with even degree $\forall v \in V$ with smallest number of edges s.t. FTSOC G is not Eulerian. Consider trail C starting and ending at v, visiting nodes through distinct edges until can't proceed anymore. Delete this closed trail decreases each vertex degree by an even number, but $G' = G \setminus C$ now has add vertices of even degree. G' is smaller than G, each CC of G' has an Euleerian tour, so combine with C to get Eulerian tour of G. Contradiction. $\square$

3. **Theorem**: Let G be ***undirected*** connected graph. Then G has an ***Eulerian trail*** iff every vertex except two vertices have even degree.

4. **Find Eulerian tour/trail algorithm**: assume G connected, all vertices even degrees. Start tour at some vertex v. Let $G' = G \setminus C$. Now $\forall v' \in V' : \deg(v')$ even, recursively find Eulerian tours C'_i in every CC of G'.

   Combine tours $C \cup C'_i$. Since G is connected, both must share a vertex u. Traverse tour C until reach vertex u, then traverse entire tour C'_i starting and returning to at/to u Continue traversal of C from u. $O(|V| + |E|)$.

5. **Theorem**: Let G be ***directed*** connected graph. Then G has ***Eulerian tour*** iff $\forall v \in V$: in-deg(v)=out-deg(v).

6. **Hamiltonian Cycle** visits each vertex exactly once. G is **Hamiltonian** if it admits such cycle.

7. **Ore Theorem**: Let G be connected graph on $n \geq 3$ vertices. If non-adjacent $\forall u, v \in V : d(u) + d(v) \geq n$ then G is Hamiltonian.

   Proof by contradiction: Let $n \geq 3$ be smallest number for which the claim doesn't hold. FTSOC let G be non-Hamiltonian graph on n vertices with maximal number of edges which satisfies theorem conditions.

   Then $\exists$ pair of non-adjacent vertices connected by a Hamiltonian path in G. If G was complete graph on $n \geq 3$ vertices, then it had Hamiltonian cycle, so G isn't complete, adding any edge will make it Hamiltonian.

   Let $u_1, ..u_n$ be such path. Since u_1, u_n are non-adjacent in G, then $d(u_1) + d(u_n) \geq n$. By pigeon-hole principle, there exists index $i \in \{2, 3, ..n-1\}$ s.t u_1 is adjacent to u_i and u_n is adjacent to u_{i-1}.

   Then $u_i - u_1 - .. - u_{i-1} - u_n - u_{n-1} - .. - u_i$ is Hamiltonian cycle. Contradiction. $\square$

8. **Dirac Theorem**: Let G be a graph on $n \geq 3$ vertices. If $\min_{v \in V} d(v) \geq n/2$ then G is Hamiltonian.

   Proof follows from Ore's Theorem. $\square$