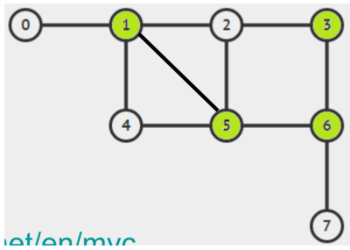
COMBINATORIAL OPTIMIZATION PROBLEMS

VERTEX COVER

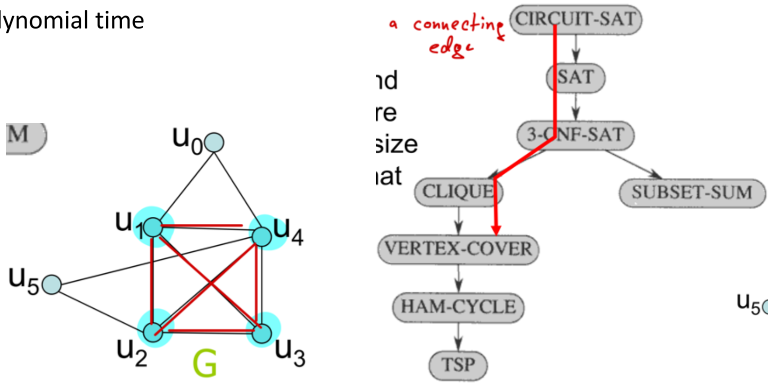
DEFINITION



Vertex Cover of Graph G = (V, E): subset S ⊆ V such that **for every e = (u, v) ∈ E, u ∈ S or v ∈ S**

PROOF NP-COMPLETE

NP = can be solved in polynomial time by a non-deterministic machine and verified in polynomial time by a deterministic machine
NP-Hard = every problem in NP is reducible to L in polynomial time
NP-Complete = NP && NP-Hard



1. Vertex-Cover in NP: (verify in polynomial time)

Input: An undirected graph **G = (V, E)** and an integer **k**
Certificate: A subset **V'** of size **k**

The **O(V+E)** verification algorithm checks:
- if **|V'| = k** and insert those vertices into a hash table (**O(1)** per that data structure insertion, so **O(V)** overall)
- Then, it scans all edge **(u, v)** ∈ **E** to check if at least one of **u** and **v** belongs to **V'** (**O(1)** per that data structure check, so **O(E)** overall)

2. Vertex-Cover in NP-Complete: (polynomial time reduction) **CLIQUE$\leq_p$VERTEX-COVER**
Clique of Graph G = (V, E): subset C ⊆ V such that every pair of vertices in C has a connecting edge
→ reduce Clique-Problem (NP-Hard) into Vertex Cover

Given an undirected graph **G = (V, E)** and **k** (note: **n = |V|**), we construct a graph **$\bar{G}$ = (V, $\bar{E}$)** where **(u, v)** ∈ **$\bar{E}$** iff **(u, v)** ∉ **E**

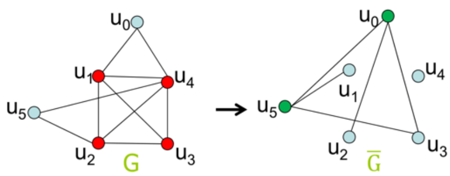
Claim: G has a size-k clique iff $\bar{G}$ has a size-(n-k) vertex cover

Conversely, suppose U ⊆ V is the size-(**n-k**) vertex cover of $\bar{G}$,
e.g., U = {u_0, u_5} is a size-**2** vertex cover of $\bar{G}$
- By definition of vertex cover, for all u, v ∈ V, if (u, v) ∈ $\bar{E}$, then at least one of u and v belong to U
- The contrapositive of this statement is for all u, v ∈ V and both u and v do not belong to U → (u, v) ∉ $\bar{E}$ → (u, v) ∈ E

Hence, V-U is a clique and V-U has size = **n-(n-k) = k**,
e.g., V-U = {u_1, u_2, u_3, u_4} is a size-**4** clique of G

Suppose V' ⊆ V is a size-**k** clique of G,
e.g., V' = {u_1, u_2, u_3, u_4} is a size-**4** clique of G
- For any arbitrary edge (u, v) ∈ $\bar{E}$ in the complement graph $\bar{G}$, then (u, v) ∉ E
  - Which implies that at least one of u or v does not belong to a clique V' as **every pair** of vertices in V' are connected by an edge in E and thus won't be in $\bar{E}$

Hence, at least one of u and v belongs to V-V' (of size **n-k**),
e.g., V-V' = {u_0, u_5} is a size-**2** vertex cover of $\bar{G}$
- Since edge (u, v) ∈ $\bar{E}$ was chosen arbitrarily, every edge (u, v) ∈ $\bar{E}$ is covered by a vertex in V-V', so V-V' (of size **n-k**) is a VC of $\bar{G}$



MVC is NP-hard: O(1) polynomial reduction VC <=$_p$ MVC (check if min <= k)
MVC is not NP-complete: no polynomial verifier (have to try every possible cover)

→ MVC solves MIS (Max-Independent-Set: no connecting edges between vertices) → complement V \ MVC

Fast, optimal, universal → special case, parameterized solution (assume), approximate solution

MVC ON TREE (DP)

- $in(v) = 1 + \Sigma_{c \subset children(v)} \min(in(c), out(c))$
- $out(v) = \Sigma_{c \subset children(v)} in(c)$
- Base case at a leaf v: $in(v) = 1$, $out(v) = 0$
- answer = $\min(in(r), out(r))$, computable in **O(n)**

No Cycles, small graph size

Only 2xV States, at most two incoming edges -> O(V)

MVC solvable in polynomial (also if no odd cycles)

PARAMETERIZED MVC

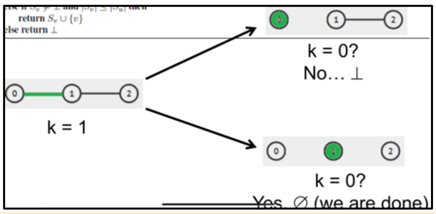
Pure Brute Force: $O(2^n m)$ → **NO Cycles**

Parameterized: k << n → Naive $O(n^k m)$ algorithm or $O(2^k m)$

```
Algorithm: ParameterizedVertexCover(G = (V, E), k)
Procedure:
   /* Base case of recursion:  ⊥ = failure, ∅ = empty set
   if k = 0 and E ≠ ∅ then return ⊥
   if E = ∅ then return ∅
   /* Recurse on arbitrary edge e:
   Let e = (u, v) be any edge in G.
   S_u = ParameterizedVertexCover(G_{-u}, k − 1)
   S_v = ParameterizedVertexCover(G_{-v}, k − 1)
   if S_u ≠ ⊥ and |S_u| < |S_v| then
      return S_u ∪ {u}
   else if S_v ≠ ⊥ and |S_v| ≤ |S_u| then
      return S_v ∪ {v}
   else return ⊥
```

T(k, m) <= 2 T(k-1, m) + O(m)

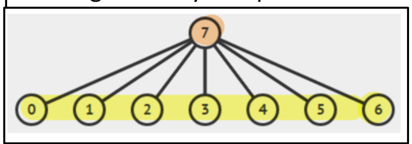
$O(2^k m)$



APPROX MVC

```
/* This algorithm adds vertices greedily, one at a time, until everything
   is covered.  At each step, the algorithm chooses the next vertex that
   will cover the most uncovered edges.                                  */
Algorithm: ApproxVertexCover-3(G = (V, E))
Procedure:
   C ← ∅
   /* Repeat until every edge is covered:                                */
   while E ≠ ∅ do
      Let d(x) = number of uncovered edges adjacent to x.
      Let u = argmax_{x∈V} d(x)
      C ← C ∪ {u}
      G ← G_{-{u}} // Remove u and all adjacent edges from G.
   return C
```

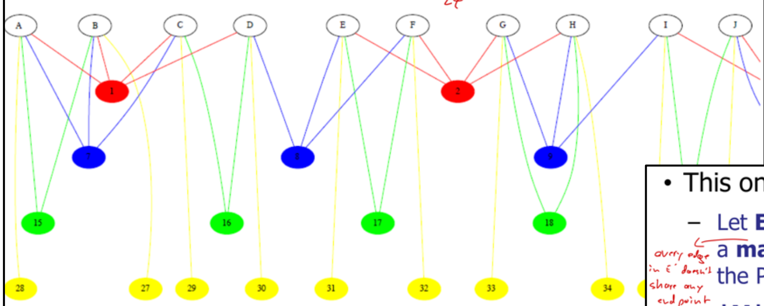
adding arbitrary end point:



Randomized: P(right choice) = 0.5 → C at most 2 * OPT (expeced) vs. Star graph

Optimal MVC = n! = 4! whites

24



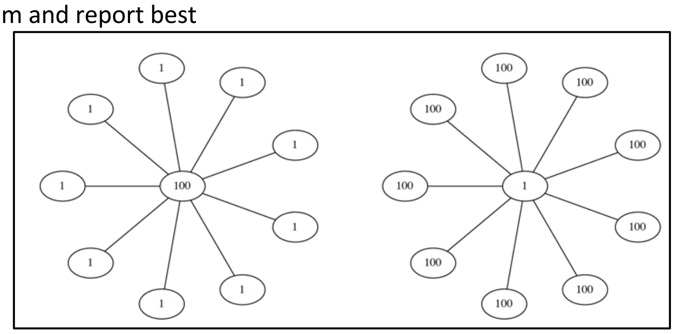
The output of ApproxVertexCover-3 *could be*:
n!/n reds + n!/3 blues + n!/2 greens + n!/1 yellows =
n! * (1/n+1/3+1/2+1/1) = n! * (1/1+1/2+1/3+1/n) = n! * ln n
(Harmonic series)
This is ln-n (or log-n, base of log is negligible) factor worse than optimal answer

Deterministic 2: adding both ends (only this one is 2-Approximation)

Deterministic 3: O(log n) approx.

- This one has 2-approximation proof
  - Let **E'** be the edges considered by this algorithm, this **E'** is a **matching M** (revisited by Week 06 of CS4234, details in the PDF), **E' = M**
    every edge in E' doesn't share any end point
  - **|M| ≤ |OPT(G)|** (details in the PDF) and **C = 2*|E'|**
  - So **C = 2*|E'| = 2*|M| ≤ 2*|OPT(G)|**

All run in polynomial time (fast) → run all of them and report best



MIN SET COVER

Set Cover of X = {x_1, x_2, … x_n} with subsets S_1, S_2, … S_m: set I $\subseteq$ {1, 2, … m} such that $\cup_{j\in I} S_j = X$

VC <=$_p$ SC → SC is NP-Hard

GREEDY SET COVER

```
/* This algorithm adds sets greedily, one at a time, until everything is
   covered.  At each step, the algorithm chooses the next set that will
   cover the most uncovered elements.
```
Algorithm: GreedySetCover($X, S_1, S_2, \ldots, S_m$)

Procedure:

$I \leftarrow \emptyset$

/* Repeat until every element in X is covered:

while $X \neq \emptyset$ **do**

Let $d(j) = |S_j \cap X|$ // This is the number of uncovered elements in S_j

Let $j = \mathbf{argmax}_{i\in\{1,2,\ldots,m\}} d(i)$ // Break ties by taking lower i

$I \leftarrow I \cup \{j\}$ // Include set S_j into the set cover

$X \leftarrow X \setminus S_j$ // Remove elements in S_j from X.

return I

O(log n) Approximation:

When we run the algorithm, let us label the elements in the order that they are covered.

$$\underbrace{x_1, x_2, x_3, x_4, x_5, x_6}_{S_2}, \underbrace{x_7, x_8, x_9}_{S_3}, \underbrace{x_{10}, x_{11}}_{S_4}, \underbrace{x_{12}}_{S_1}$$

← These are the first sets that cover these elements

For each element x_j, let c_j be the number of elements covered at the same time. In the example, this would yield:

$$c_1 = 6, c_2 = 6, c_3 = 6, c_4 = 6, c_5 = 6, c_6 = 6, c_7 = 3, c_8 = 3, c_9 = 3, c_{10} = 2, c_{11} = 2, c_{12} = 1$$

We define $cost(x_j) = 1/c_j$. In this way, the cost of covering all the new elements for some set is exactly 1. In this example, the cost of covering $x_1, x_2, x_3, x_4, x_5, x_6$ is 1, the cost of covering x_7, x_8, x_9 is 1, etc. In general, if I is the set cover constructed by the Greedy Algorithm, then:

$$|I| = \sum_{j=1}^{n} cost(x_j) .$$

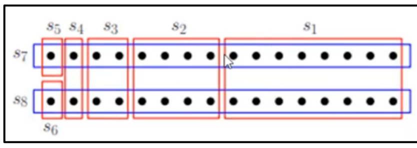
Therefore, OPT needs at least $(n-j+1)/c(j)$ sets to cover the remaining $(n-j+1)$ elements. We thus conclude that:

$$OPT \geq \frac{n-j+1}{c(j)} \geq (n-j+1)cost(x_j)$$

Or to put it differently:

$$cost(x_j) \leq \frac{OPT}{(n-j+1)}$$

$$\begin{aligned} |I| &= \sum_{j=1}^{n} cost(x_j) \\ &\leq \sum_{j=1}^{n} \frac{OPT}{(n-j+1)} \\ &\leq OPT \sum_{i=1}^{n} \frac{1}{i} \\ &\leq OPT(\ln n + O(1)) \end{aligned}$$



Opt = {s_7, s_8}

Greedy: $\{s_1, s_2, s_3, s_4, s_5, s_6\}$

3 times more subsets than optimal answer

STEINER TREE

EUCLIDEAN-STEINER-TREE

Definition: given set R of n 2D-points in **Euclidean plane**, find set of additional Steiner points S and spanning Tree T = (R $\cup$ S, E) such that weight of tree is minimized (NP-Hard) $\sum_{(u,v)\in E} |u - v|$

- Each Steiner Point has **degree 3**

- Lines form **120 degree** angles
- At most **n-2 Steiner points**

METRIC-STEINER-TREE

$$\sum_{(u,v)\in E} d(u,v) .$$

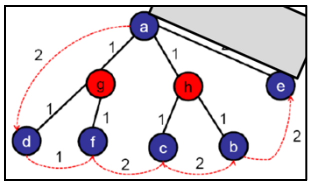
Given required vertices R, set of Steiner vertices S, distance function d find subset $S' \subset S$ and spanning tree $T = (R \cup S', E)$ of min weight

Metric easier than Euclidean (criterion on how many Steiner points needed and where to place them)
Generalized: given an arbitrary graph with edge weights

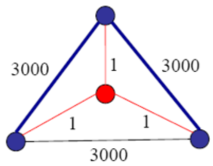
COMPLETE SEARCH SOLUTION

Try all possible subsets of Steiner vertices ($2^{|n-s|}$) → run MST algo O(E log V) → $O(2^{|n-s|} * n^2 * \log n)$

METRIC MST 2-APPROXIMATION (NO STEINER POINT)



Create a cycle bypassing all Steiner vertices and then removing duplicate vertices (generate short-cutting paths, triangle inequality) → break one edge to obtain acyclic spanning tree
cost(M) <= cost(C) <= 2*cost(T) with cost(T) = cost of DFS



GENERAL MST 2-APPROXIMATION (GAP-PRESERVED)

Metric completion: non-metric edge weights into metric ones with All-Pairs Shortest Path algorithm (e.g. Floyd-Warshall $O(V^3)$) → preserves metric properties *(e.g., proof triangle inequality by contradiction)*

Definition 7 *Given a graph $G = (V, E)$ and non-negative edge weights w, we define the metric completion of G to the be the distance function $d : V \times V \to \mathbb{R}$ constructed as follows: For every $u, v \in V$, define $d(u, v)$ to be the distance of the* shortest path *from u to v in G with respect to the weight function w.*

Convert back to General-ST: replace virtual edges in Metric-ST with the actual shortest paths, remove overlapping edges and cycles → cost equal or lower in General ST-version

Use **Theorem:** given an α-approx. algo. for finding a metric ST, we can find an α-approx. algo for a general ST
Gap-preserving: reduction that preserves approximation ratios

TRAVELING SALESMAN (TSP)

	Repeats	No-Repeats
Metric	M-R-TSP	M-NR-TSP
General	G-R-TSP	G-NR-TSP

Definition: given a set V of n points and a distance function d, find cycle C of minimum cost that contains all points; Complete Graph $O(V^2)$
Botanic Variant: solvable in $O(N^2)$

4 Variants (3 equivalent), all NP-hard (G-NR-TSP even NP-hard to approx)

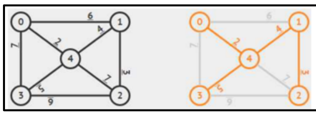
BRUTE FORCE & DP

(N-1)! Permutations if we fix one node → O(N! * N) time, O(N) dist. sum calc.
Improvement: memorize repeated sub-tours $O(N^2 * 2^{N-1})$, Held-Karp DP for TSP

→ small graph, TSP is bitonic, each vertex visited exactly once

2-APPROX FOR G-R-TSP

Run MST of input graph → Run DFS on resulting MST → Output vertices in cycle induced by DFS (no repeat)



DFS={0,4,1,2,1,4,3,4,0}
Output C={0,4,1,2,3,0}

Proof:

- C* = OPT(V, d); E* = edges in optimal cycle; T* is MST of G = (V, E*) → d(T*) <= d(C*) = OPT
- T = MST of G = (V, E) → E* ⊆ E → d(T) <= d(T*)
  → d(C) = 2 * d(T) //see Metric ST Analysis <= 2 * d(T*) <= 2 * d(C*) <= 2* OPT

1.5-APPROX FOR M-NR-TSP

Eulerian Cycle: a cycle that crosses each edge exactly once (connected + every vertex even degree)
Perfect Matching: subset M of edges in a graph so that no two edges share an endpoint; perfect: |M| = |V|/2

Christofides's Algorithm:

1. T = MST(G) and E = edges of T
2. O = set of vertices with odd degree (even number, Handshaking lemma)
3. M = Min-Weight-Perfect-Matching on subgraph G* induced by O
4. Combine T+M (all vertices have even degree)
5. Output vertices in Eulerian Cycle (no repeat)

Analysis:

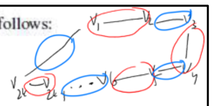
Cost(E) = $\sum_{e \in E}$ d(e) + $\sum_{e \in M}$ d(e)
(all edges in MST) + (all edges in Min-Weight-Perfect-Matching)

$\sum_{e \in E}$ d(e) ≤ OPT
using the now-classic technique: OPT is a TSP cycle of graph G = (V, E), if we remove any edge from this cycle, we will get a spanning tree and the MST of the graph G must have cost no greater than this cycle (usually smaller, as we delete at least one positive weighted edge)

- Cycle **C' on odd vertices** (even, with no repeats): **cost (C') <= cost(C)**, as we only skipped vertices (triangle inequality)
- M = min cost perfect matching
  - Cost(M) <= cost(M1)
  - Cost(M) <= cost(M2) (each has |O|/2 edges)
- 2 cost(M) <= cost(m1) + cost(M2) <= cost(C') <= cost(C) = OPT
- Cost(M) <= OPT/2

construct two different perfect matchings M_1 and M_2. We define them as follows:

$$M_1 = (v_1, v_2), (v_3, v_4), (v_5, v_6), \dots, (v_{2k-1}, v_{2k})$$
$$M_2 = (v_2, v_3), (v_4, v_5), (v_6, v_7), \dots, (v_{2k}, v_1)$$



Cost(E) = $\sum_{e \in E}$ d(e) + $\sum_{e \in M}$ d(e)
≤ cost(T) + cost(M)
≤ OPT + OPT/2
≤ 1.5 OPT

LINEAR PROGRAMMING

LPS

SIMPLEX METHOD (LP: POLYNOMIAL TIME)

1. Find any (feasible) vertex v (normally (0,0))
2. Examine all the neighboring vertices of v: $v_1, v_2, \ldots, v_k$
3. Calculate $f(v), f(v_1), f(v_2), \ldots, f(v_k)$
   – If f(v) is the maximum (among its neighbors), then stop and return v
4. Otherwise, choose **<u>one of</u>** the neighboring vertices v_j where $f(v_j) > f(v)$
   – Let $v = v_j$
5. Go to step (2)

Details of Simplex is "dropped" in

Vertex = intersection of some constraints

If equations are not linearly separable you will get many solutions

m constraints, n variables → mCn = $O(m^n)$

MWVC AS ILP

$$\min \left(\sum_{j=1}^{n} w(v_j) \cdot x_j \right) \quad \text{where:}$$

$$\begin{aligned} x_i + x_j &\geq 1 && \text{for all } (i,j) \in E \\ x_j &\geq 0 && \text{for all } j \in V \\ x_j &\leq 1 && \text{for all } j \in V \\ x_j &\in \mathbb{Z} && \text{for all } j \in V \end{aligned}$$

weighted one

→ MVC <=(p) ILP → ILP is also np-hard

RELAXATIONS (ILP: NP-HARD)

No integer constraint → round up x if it is <= 0.5

2-Approximation algorithm:

→ searching on bigger space

OPT(G) = OPT(ILP) ≥ OPT(LP)

$$cost(OPT) \geq \sum_{j=1}^{n} w(v_j) \cdot x_j$$

Assume w is all 1
Example ILP solution x_0
Example LP solution x_0

Rounded answer ≤ 2 x OPT(LP)

$$\begin{aligned} \sum_{j=1}^{n} w(v_j) \cdot \boxed{y_j} &\leq \sum_{j=1}^{n} w(v_j) \cdot \boxed{(2x_j)} \\ &\leq 2\left(\sum_{j=1}^{n} w(v_j) \cdot x_j\right) \\ &\leq 2 \times OPT(G) \end{aligned}$$

Notice, however, that $y_j \leq 2x_j$, f

FLOWS & MATCHING

MAX FLOW – NOT NP HARD!

st-cut: partitions vertices of a graph into 2 disjoint sets S and T (source s ∈ S, sink t ∈ T)
cap of st-cut: sum of capacities of edges that cross cut **from S to T**
net-flow: flow on edges from S->T minus flow from T->S

flow f = net-flow of any st-cut <= cap of st-cut (**Weak duality**)

Induction-Proof: start with S = {s}, T= V\S → take node x, add/subtract outgoing/incomding edges, subtract/add edges from X to S → flow into X = flow out of X → F unchanged → same for all cuts

MAXFLOW-MINCUT THEOREM

f is max flow ⇔ cut whose capacity equals value of f (min cap) ⇔ no augmenting paths in residual graph

- 2 -> 1: st-cut with min cap → for all flows g: value(g) <= cap(S, T) = value(f) → f is max flow
- 1 -> 3: assume 1 more augmenting path → send one flow and improve flow → f not max flow → contradiction
- 3 -> 1: source cannot reach sink anymore

FORD-FULKERSON

Idea: find augmenting path (from s to t through edges with residual capacity left) along which flow++

If Ford-Fulkerson terminates there is no augmenting path left → flow is max

FF always terminates (if cap integers):

- Every iteration finds new augmenting path → bottleneck cap of at least 1
- Each iteration increases flow of at least one edge by at least 1
- Finite number of edges, finite max cap per edge → termination

Complexity: **O(m^2U)**

- O(m) for finding path p in R and updating caps (m >> n)
- U = max cap of outgoing edge connected to s → MF <= m*U

Ford-Fulkerson Algorithm

Start with 0 flow.

Build residual graph:

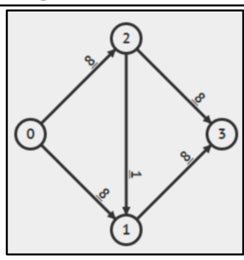
- For every edge (u,v) add edge (u,v) with w(u,v) = capacity.
- For every edge (u,v) add (a new) edge (v,u) with w(v,u) = 0.

Be care
of poter
bug(s) h

While there exists an augmenting path:

- Find an augmenting path **via DFS** (**the 'wrong one first'**) in residual graph.
- Compute bottleneck capacity.
- Increase flow on the path by the bottleneck capacity:
  - For every edge (u,v) on the path, subtract the flow from w(u,v).
  - For every edge (~~u,v~~ v,u) on the path, add the flow to w(v,u).

Compute final flow by inverting residual flows.



EDMONDS KARP

Run O(E) BFS to find the shortest (in terms of edges used) augmenting path
Complexity: O(m^2n) → strongly polynomial algorithm → NOT NP-hard

DINIC

Uses BFS information in a better way than Edmonds-Karp (90% identical) → O(V^2*E)

FINDING EDGES IN MIN-CUT

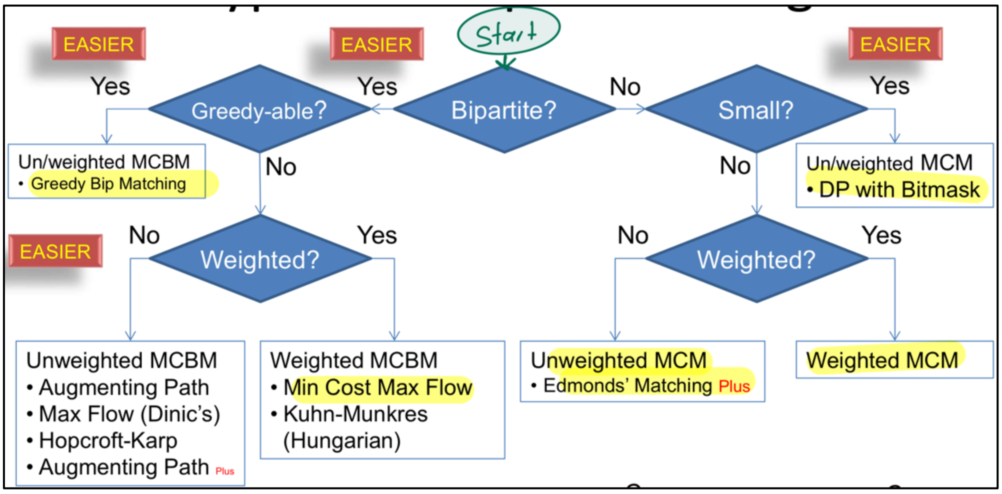
Run Maxflow algo until termination → find vertices S that are still reachable from source (DFS) → T = V\S → for every edge in S: if endpoint in T add to min-cut

(WEIGHTED) MAX-CARDINANLITY (BIPARTITE) MATCHING

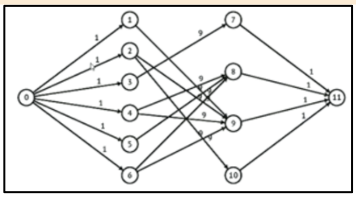
*MCBM Keywords: Left/Right, Row/Col, alternate Row/Col, Prime/Coprime, Odd/even, male/female, job/employee, bi-coloring, out/in-degree only, no-odd-length cycle, Tree**

Matching: subset M of edges in a graph G = (V, E) so that no two edges share an endpoint

Bipartite: vertices partitioned into 2 disjoint sets U and V, such that every edge can only connect from U to V



MCBM BY REDUCING INTO MAXFLOW



Directed, bipartite!, O(sqrt(V)*E) for Dinic

AUGMENTING PATH ALGORITHM

Berges theorem:

- Matching M is maximum if and only if there is no augmenting path with M
- Augmenting path: starts and ends on unmatched vertices and alternates between edges in and not in the matching

Proof:

- Max → no augmenting path:
  Contradiction: 1 more augmenting path -> flip it -> get one more matching -> not max
- No Augmenting -> Max: suppose not max → M' > M → Symmetric difference (edges that is not covered by both) → consists of paths or cycles (degree <= 2)
  - Even length path/cycle → |M| = |M'| -> M' not > M
  - Odd length cycle not possible (triangle graph: cannot assign last edge)

- Odd length path: starts with edges from larger M’ and edges in M are inside → aug path → contradiction

```
vi match, vis;                                    // global variables
vector<vi> AL;

int Aug(int L) {
  if (vis[L]) return 0;                           // L visited, return 0
  vis[L] = 1;
  for (auto &R : AL[L])
    if ((match[R] == -1) || Aug(match[R])) {
      match[R] = L;                               // flip status
      return 1;                                   // found 1 matching
    }
  return 0;                                       // no matching
}
```

For each vertex in the left:
- if there is an augmenting path of 1+ edges -> flip edge status along path

O(VE),

Weakness: for very connected graphs augmenting paths will be very long in the last stages → randomly O(V+E) select neighbour → O(V^2+kE)

Maxflow: for variation (use left or right multiple times), multiple layers

HOPCROFT KARP (HK)

Identical to Dinic Max Flow → prioritize shortest augmenting paths (number of edges) → O(sqrt(V)*E)

HALL’S MARRIAGE THEOREM

Bipartite Graph with sets U and V → a matching covering U exists if and only if for each subset W of U: |W| <= |N(W)| → 2^W checks required

RANDOM

DAG = Directed acyclic graph
n <= 25 is upper limit of what $O(2^n)$ algorithm can do in 1s

MST

Prim, Kruskal: O(V^2 log V)

DFS, BFS

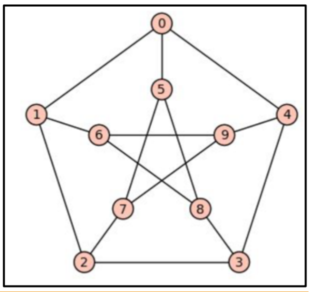
ITERATIVE BRUTE FORCE COPS

$O(2^V)$ and use Bitmask (normally V > 10 is too large

METRIC

- Non-negativity: *For all* $u, v \in V$, $d(u, v) \geq 0$.
- Identity: *For all* $u \in V$, $d(u, u) = 0$.
- Symmetric: *For all* $u, v \in V$, $d(u, v) = d(v, u)$.
- Triangle inequality: *For all* $u, v, w \in V$, $d(u, v) + d(v, w) \geq d(u, w)$.

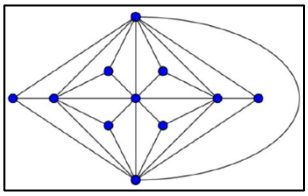
PLANAR GRAPH CRITERIA



Kuratowski & Wagner: A graph is planar if and only if it does not contain K5 and K(3, 3) minor → Biggest Clique a planar graph can have is of size 4

Number of edges <= 3n-6

4-Colour-Theorem: in any planar graph you need at most 4 color to color the graph



LIST OF NP-HARD COPS

1. `Min-Vertex-Cover` (+weighted version, Lec1+Lec2)
2. `Max-Clique` (mentioned briefly and in Tut01)
3. `Graph-Coloring` (mentioned briefly in Tut01)
4. `Min-Set-Cover` (+weighted version, Lec3)
5. `Steiner-Tree` (3 variants, Lec3)
6. `Min-Feedback-Edge-Set` (+weighted version, Tut02)
7. `Partition` (+weighted version, Tut02)
8. `Travelling-Salesman-Problem` (4 variants, Lec4)
9. `Max-Independent-Set` (Tut03)