21-0: **Classes of Problems**

- Consider three problem classes:
  - Polynomial (P)
  - Nondeterminisitic Polynomial (NP)
  - NP-Complete
- (only scratch the surface, take Automata Theory to go in depth)

21-1: **Class P**

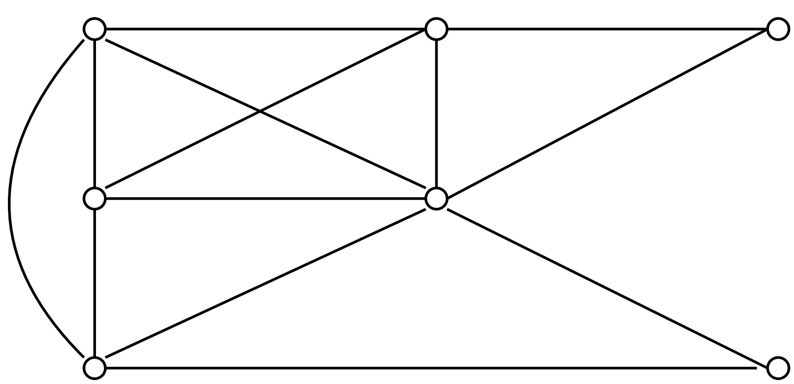
- Given a problem, we can find a solution in polynomial time
  - Time is polynomial in the length of the problem description
  - Encode the problem in some resonable way (like a string S)
  - Can create a solution to the problem in time $O(|S|^k)$, for some constant k.

21-2: **Class P Example**

- Reachability
- Given a Graph G, and two vertices x and y, is there a path from x to y in G?
  - Encode the graph as an adjacency list
  - Can solve the problem in polynomial time
  - DFS

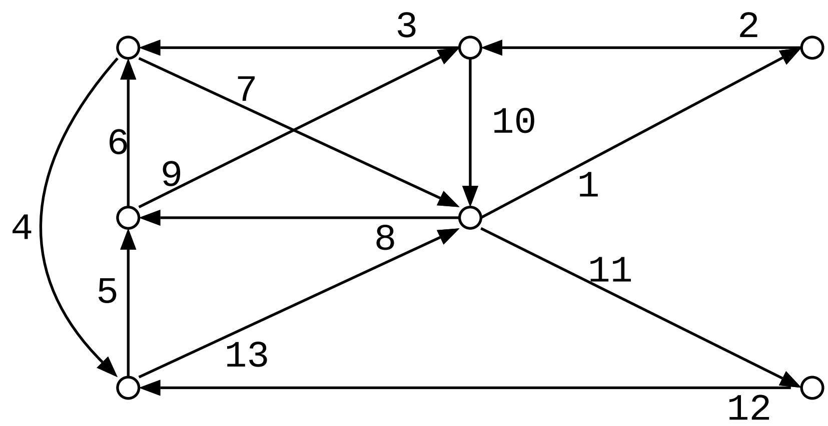
21-3: **Euler Cycles**

- Given an undirected graph G, is there a cycle that traverses every edge exactly once?



21-4: **Euler Cycles**

- Given an undirected graph G, is there a cycle that traverses every edge exactly once?



21-5: **Euler Cycles**

- We can determine if a graph G has an Euler cycle in polynomial time.
- A graph G has an Euler cycle if and only if:
  - G is connected
  - All vertices in G have an even # of adjacent edges

21-6: **Euler Cycles**

- Pick any vertex, start following edges (only following an edge once) until you reach a "dead end" (no untraversed edges from the current node).
- Must be back at the node you started with
  - Why?
- Pick a new node with untraversed edges, create a new cycle, and splice it in
- Repeat until all edges have been traversed

21-7: **Class P Example**

- *Almost* every algorithm we've seen so far has been in P.
  - *Possible* exception: Knapsack problem
- If a problem is not in P, it takes exponential time to solve
  - Not practical for large problems

21-8: **NP**

- Nondeterministic Polynomial (NP) problems:
  - Given a solution, that solution Can be verified in polynomial time
  - If we could guess a solution to the problem (that's the Non-deterministic part), we could verify the solution quickly (polynomial time)
  - All problems in P are also in NP

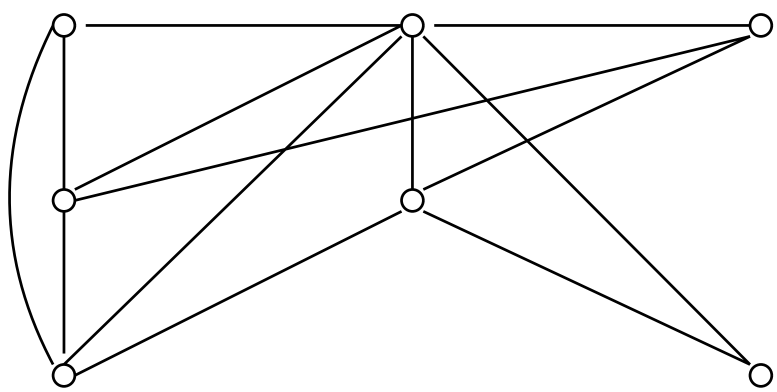
- Most problems are in NP
-

21-9: **NP – Example**

- Reachability is also in NP
- Given a Graph G, and two vertices x and y, is there a path from x to y in G?
- Given a graph G and two verticies x and y, we can determine if the path does in fact connect x and y ing G, in polynomial time
  - Make sure each edge in the path exists in the graph
- All problems in P are also in NP

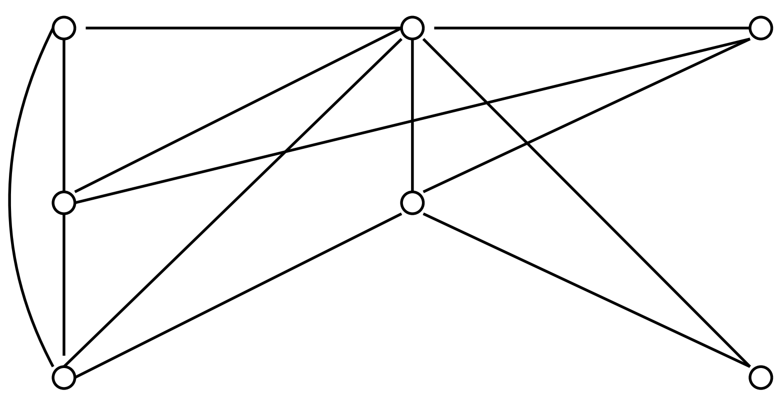
21-10: **Hamiltonian Cycles**

- Given an undirected graph G, is there a cycle that visits every vertex exactly once?



21-11: **Hamiltonian Cycles**

- Given an undirected graph G, is there a cycle that visits every vertex exactly once?



21-12: **Hamiltonian Cycles**

- Given an undirected graph G, is there a cycle that visits every vertex exactly once?
  - Very similar to the Euler Cycle problem

- Verifyable in polynomial time
- No known polynomial time solution

21-13: **Satisfiability**

- A Boolean Formula in Conjunctive Normal Form (CNF) is a conjunction of disjunctions.
  - $(x_1 \vee x_2) \wedge (x_3 \vee \overline{x_2} \vee \overline{x_1}) \wedge (x_5)$
  - $(x_3 \vee x_1 \vee x_5) \wedge (x_1 \vee \overline{x_5} \vee \overline{x_3}) \wedge (x_5)$
- A Clause is a group of variables x_i (or negated variables $\overline{x_j}$) connected by ORs ($\vee$)
- A Formula is a group of clauses, connected by ANDs ($\wedge$)

21-14: **Satisfiability**

- Satisfiability Problem: Given a formula in Conjunctive Normal Form, is there a set of truth values for the variables in the formula which makes the formula true?
- $(x_1 \vee x_4) \wedge (\overline{x_2} \vee x_4) \wedge (x_3 \vee x_2) \wedge$
  $(\overline{x_1} \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_4})$
  - Satisfiable: x_1 = T, x_2 = F, x_3 = T, x_4 = F
- $(x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2)$
  - Not Satisfiable

21-15: **Class NP-Complete**

- A problem is NP-Complete if:
  - Problem is NP
  - *If* you could solve the problem in polynomial time, then you could solve *all* NP problems in polynomial time
- Reduction:
  - Given problem A, create an instance of problem B (in polynomial time)
  - Solution to problem B gives a solution to problem A
  - If we could solve B, in polynomial time, we could solve A

21-16: **Reduction Example**

- Given any instance of the Hamiltonian Cycle Problem:
  - We can (in polynomial time) create an instance of Satisfiability
  - That is, given any graph G, we can create a boolean formula f, such that f is satisfiable if and only if there is a Hamiltonian Cycle in G
- If we could solve Satisfiability in Polynomial Time, we could solve the Hamiltonian Cycle problem in Polynomial Time

21-17: **Reduction Example**

- Given a graph G with n vertices, we will create a formula with n^2 variables:
  - $x_{11}, x_{12}, x_{13}, \ldots x_{1n}$
    $x_{21}, x_{22}, x_{23}, \ldots x_{2n}$
    $\ldots$
    $x_{n1}, x_{n2}, x_{n3}, \ldots x_{nn}$
- Design our formula such that x_{ij} will be true if and only if the ith element in a Hamiltonian Circuit of G is vertex # j

21-18: **Reduction Example**

- For our set of n^2 variables x_{ij}, we need to write a formula that ensures that:
  - For each i, there is exactly one j such that x_{ij} = true
  - For each j, there is exactly one i such that x_{ij} = true
  - If x_{ij} and $x_{(i+1)k}$ are both true, then there must be a link from v_j to v_k in the graph G

21-19: **Reduction Example**

- For each i, there is exactly one j such that x_{ij} = true
  - For each i in $1 \ldots n$, add the rules:
    - $(x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$
- This ensures that for each i, there is at least one j such that x_{ij} = true
- (This adds n clauses to the formula)

21-20: **Reduction Example**

- For each i, there is exactly one j such that x_{ij} = true

```
for each i in 1 ... n
    for each j in 1 ... n
        for each k in 1 ... n        j ≠ k
            Add rule (¬x_ij ∨ ¬x_ik)
```

(where the rule is $(\overline{x_{ij}} \vee \overline{x_{ik}})$)

- This ensures that for each i, there is at most one j such that x_{ij} = true
- (this adds a total of n^3 clauses to the formula)

21-21: **Reduction Example**

- For each j, there is exactly one i such that x_{ij} = true
  - For each j in $1 \ldots n$, add the rules:
    - $(x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj})$
- This ensures that for each j, there is at least one i such that x_{ij} = true
- (This adds n clauses to the formula)

21-22: **Reduction Example**

- For each j, there is exactly one i such that x_{ij} = true

for each j in $1 \ldots n$
    for each i in $1 \ldots n$
        for each k in $1 \ldots n$
            Add rule $(\overline{x_{ij}} \vee \overline{x_{kj}})$

- This ensures that for each j, there is at most one i such that x_{ij} = true
- (This adds a total of n^3 clauses to the formula)

21-23: **Reduction Example**

- If x_{ij} and $x_{(i+1)k}$ are both true, then there must be a link from v_i to v_k in the graph G

for each i in $1 \ldots (n-1)$
    for each j in $1 \ldots n$
        for each k in $1 \ldots n$
            if edge (v_j, v_k) is *not* in the graph:
                Add rule $(\overline{x_{ij}} \vee \overline{x_{(i+1)k}})$

- (This adds no more than n^3 clauses to the formula)

21-24: **Reduction Example**

- If x_{nj} and x_{0k} are both true, then there must be a link from v_i to v_k in the graph G (looping back to finish cycle)

for each j in $1 \ldots n$
    for each k in $1 \ldots n$
        if edge (v_n, v_0) is *not* in the graph:
            Add rule $(\overline{x_{nj}} \vee \overline{x_{0k}})$

- (This adds no more than n^2 clauses to the formula)

21-25: **Reduction Example**

- In order for this formula to be satisfied:
  - For each i, there is exactly one j such that x_{ij} is true
  - For each j, there is exactly one i such that x_{ji} is true
  - if x_{ij} is true, and $x_{(i+1)k}$ is true, then there is an arc from v_j to v_k in the graph G
- Thus, the formula can only be satisfied if there is a Hamiltonian Cycle of the graph

21-26: **Proving NP-Completeness**

- Once you have the first NP-complete problem, easy to find more

- Given an NP-Complete problem P
- Different problem P'
- Polynomial-time reduction from P to P'
- P' must be NP-Complete

21-27: **Proving NP-Completeness**

- First NP-Complete problem: Satisfiability (SAT)
  - SAT is NP-Complete
  - By reduction from the universal Turing machine
  - Reduce any algorithm that guesses and verifies to SAT
  - For the actual proof, see Automata Theory
    - Main goal of the class is to build up the formal tools needed to prove SAT is NP-Complete.

21-28: **More NP-Complete Problems**

- Exact Cover Problem
  - Set of elements A
  - $F \subset 2^A$, family of subsets
  - Is there a subset of F such that each element of A appears exactly once?

21-29: **More NP-Complete Problems**

- Exact Cover Problem
  - $A = \{a, b, c, d, e, f, g\}$
  - $F = \{\{a, b, c\}, \{d, e, f\}, \{b, f, g\}, \{g\}\}$
  - Exact cover exists:
    $\{a, b, c\}, \{d, e, f\}, \{g\}$

21-30: **More NP-Complete Problems**

- Exact Cover Problem
  - $A = \{a, b, c, d, e, f, g\}$
  - $F = \{\{a, b, c\}, \{c, d, e, f\}, \{a, f, g\}, \{c\}\}$
  - No exact cover exists

21-31: **More NP-Complete Problems**

- Exact Cover is in $\mathbf{NP}$
  - Guess a cover
  - Check that each element appears exactly once
- Exact Cover is $\mathbf{NP}$-Complete
  - Reduction from Satisfiability

- Given any instance of Satisfiability, create (in polynomial time) an instance of Exact Cover

21-32: **Exact Cover is NP-Complete**

- Given an instance of SAT:
  - $C_1 = (x_1, \vee \overline{x_2})$
  - $C_2 = (\overline{x_1} \vee x_2 \vee x_3)$
  - $C_3 = (x_2)$
  - $C_4 = (\overline{x_2}, \overline{x_3})$
- Formula: $C_1 \wedge C_2 \wedge C_3 \wedge C_4$
- Create an instance of Exact Cover
  - Define a set A and family of subsets F such that there is an exact cover of A in F if and only if the formula is satisfiable

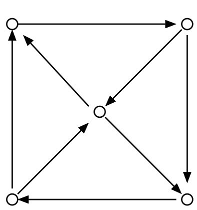
21-33: **Exact Cover is NP-Complete**

$C_1 = (x_1 \vee \overline{x_2})$ $C_2 = (\overline{x_1} \vee x_2 \vee x_3)$ $C_3 = (x_2)$ $C_4 = (\overline{x_2} \vee \overline{x_3})$

$A = \{x_1, x_2, x_3, C_1, C_2, C_3, C_4, p_{11}, p_{12}, p_{21}, p_{22}, p_{23}, p_{31}, p_{41}, p_{42}\}$
$F = \{\{p_{11}\}, \{p_{12}\}, \{p_{21}\}, \{p_{22}\}, \{p_{23}\}, \{p_{31}\}, \{p_{41}\}, \{p_{42}\},$
$X_1, f = \{x_1, p_{11}\}$
$X_1, t = \{x_1, p_{21}\}$
$X_2, f = \{x_2, p_{22}, p_{31}\}$
$X_2, t = \{x_2, p_{12}, p_{41}\}$
$X_3, f = \{x_3, p_{23}\}$
$X_3, t = \{x_3, p_{42}\}$
$\{C_1, p_{11}\}$, $\{C_1, p_{12}\}$, $\{C_2, p_{21}\}$, $\{C_2, p_{22}\}$, $\{C_2, p_{23}\}$, $\{C_3, p_{31}\}$, $\{C_4, p_{41}\}$, $\{C_4, p_{422}\}\}$

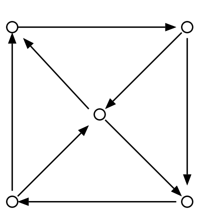
21-34: **Directed Hamiltonian Cycle**

- Given any directed graph G, determine if G has a a Hamiltonian Cycle
  - Cycle that includes every node in the graph exactly once, following the direction of the arrows



21-35: **Directed Hamiltonian Cycle**

- Given any directed graph G, determine if G has a a Hamiltonian Cycle
  - Cycle that includes every node in the graph exactly once, following the direction of the arrows

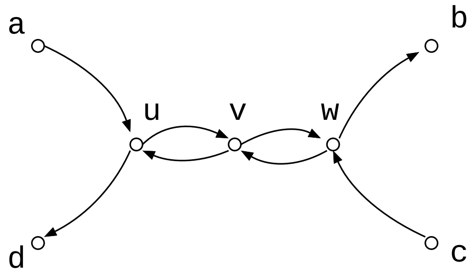


21-36: **Directed Hamiltonian Cycle**

- The Directed Hamiltonian Cycle problem is $\mathbf{NP}$-Complete
- Reduce Exact Cover to Directed Hamiltonian Cycle
  - Given any set A, and family of subsets F:
  - Create a graph G that has a hamiltonian cycle if and only if there is an exact cover of A in F

21-37: **Directed Hamiltonian Cycle**

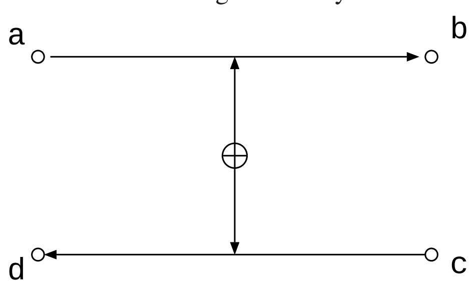
- Widgets:
  - Consider the following graph segment:



  - If a graph containing this subgraph has a Hamiltonian cycle, then the cycle must contain either $a \to u \to v \to w \to b$ or $c \to w \to v \to u \to d$ – but not both (why)?

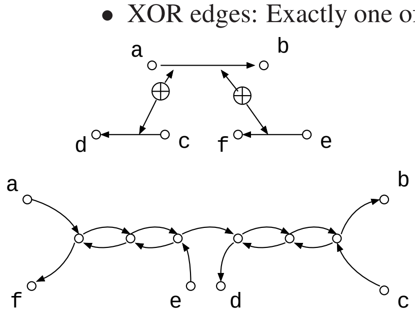
21-38: **Directed Hamiltonian Cycle**

- Widgets:
  - XOR edges: Exactly one of the edges must be used in a Hamiltonian Cycle



21-39: **Directed Hamiltonian Cycle**

- Widgets:
  - XOR edges: Exactly one of the edges must be used in a Hamiltonian Cycle



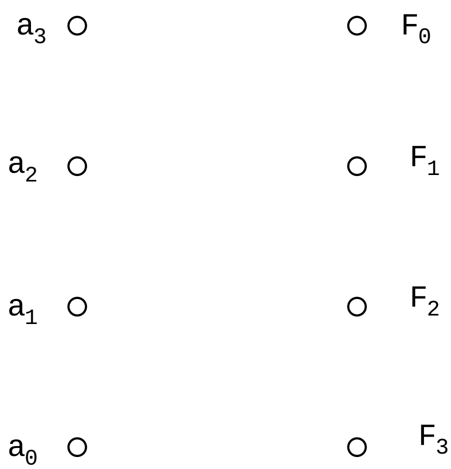
21-40: **Directed Hamiltonian Cycle**

- Add a vertex for every variable in A (+ 1 extra)

$F_1 = \{a_1, a_2\}$
$F_2 = \{a_3\}$
$F_3 = \{a_2, a_3\}$

21-41: **Directed Hamiltonian Cycle**

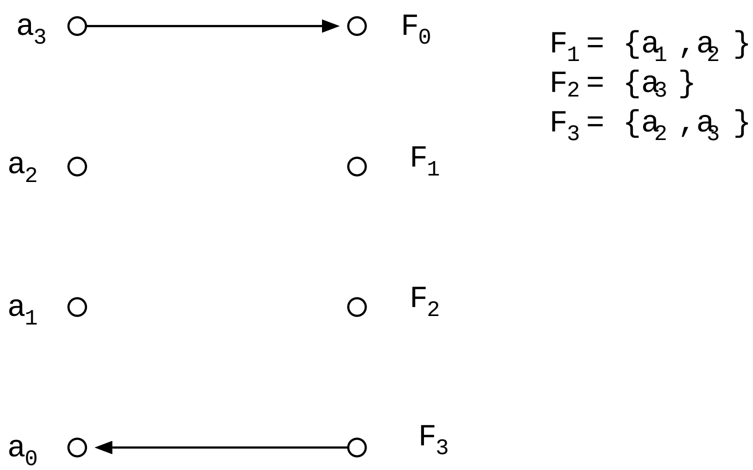
- Add a vertex for every subset F (+ 1 extra)



$F_1 = \{a_1, a_2\}$
$F_2 = \{a_3\}$
$F_3 = \{a_2, a_3\}$

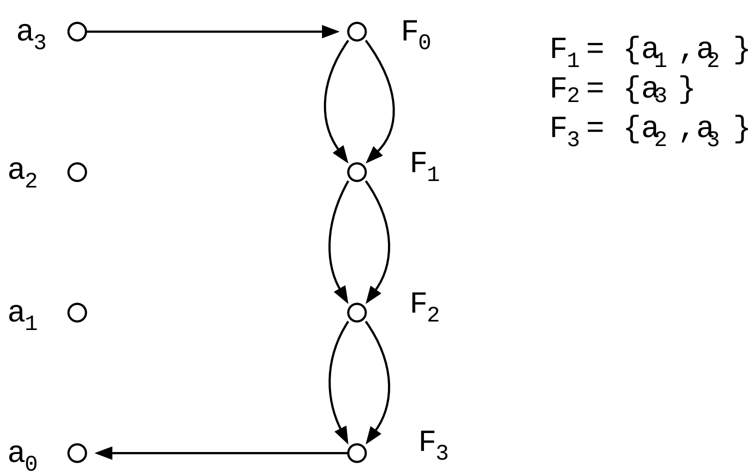
21-42: **Directed Hamiltonian Cycle**

- Add an edge from the last variable to the 0th subset, and from the last subset to the 0th variable



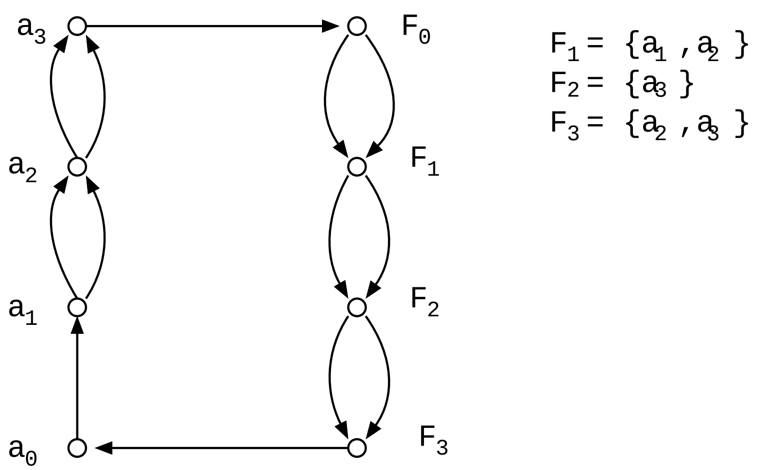
21-43: **Directed Hamiltonian Cycle**

- Add **2** edges from F_i to F_{i+1}. One edge will be a “short edge”, and one will be a “long edge”.



21-44: **Directed Hamiltonian Cycle**

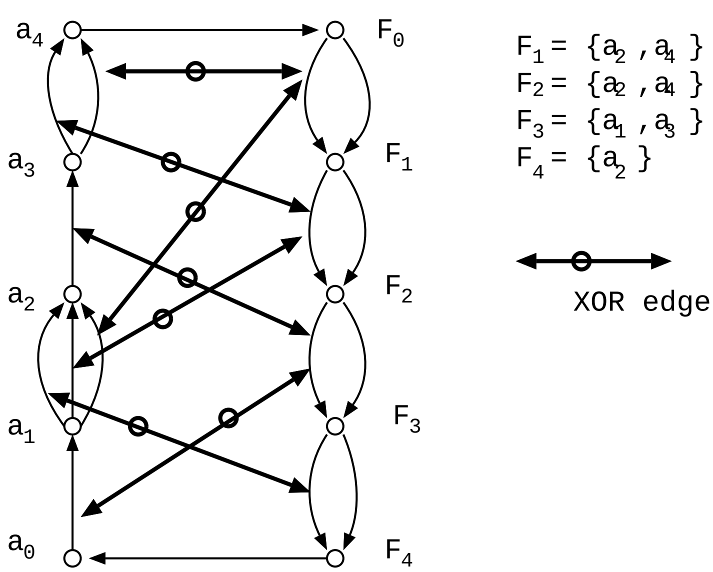
- Add an edge from a_{i-1} to a_i for **each** subset a_i appears in.



21-45: **Directed Hamiltonian Cycle**

- Each edge (a_{i-1}, a_i) corresponds to some subset that contains a_i. Add an XOR link between this edge and the long edge of the corresponding subset

21-46: **Directed Hamiltonian Cycle**



SLIDE - 21-47: **NP-Complete Problems**

- What if you need to solve an NP-Complete problem?

21-48: **NP-Complete Problems**

- What if you need to solve an NP-Complete problem?
  - If the problem is small, exponential solution is OK
  - Special case of an NP-Complete problem, that can be solved quickly (3-SAT vs. 2-SAT)
  - Approximate solution

21-49: **Approximation Ratio**

- An algorithm has an *approximation ratio* of $\rho(b)$ if, for any input size n, the cost of the solution produced by the algorithm is within a factor of $\rho(n)$ of an optimal solution

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

- For a maximization problem, $0 < C \leq C^*$
- For a minimizaion problem, $0 < C^* \leq C$

21-50: **Approximation Ratio**

- Some problems have a polynomial solution, with $\rho(n) = c$ for a small constant c.
- For other problems, best-known polynomial solutions have an approximation ration that is a function of n
  - Bigger problems $\Rightarrow$ worse approximation ratios

21-51: **Approximation Scheme**

- Some approximation algorithm takes as input both the problem, and a value $\epsilon > 0$
  - For any fixed ϵ, $(1 + \epsilon)$-approximation algorithm
  - $\rho(n) = 1 + \epsilon$
- Running time increases as ϵ decreases

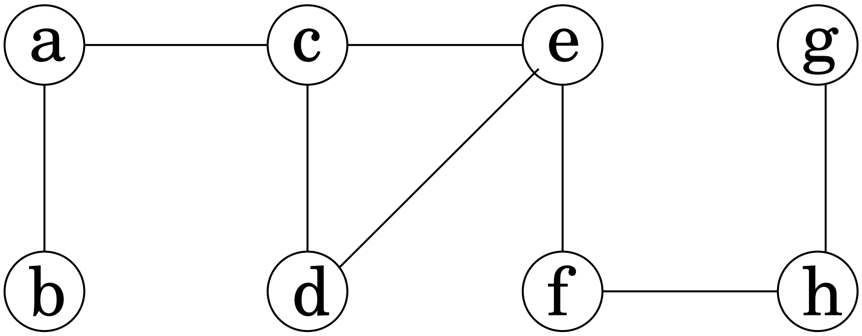
21-52: **Vertex Cover**

- Problem: Given an undirected graph $G = (V, E)$, find a $V' \subseteq V$ such that
  - For each edge (u, v) E, $u \in V'$ or $v \in V'$
  - $|V'|$ is as small as possible
- Vertex Cover is NP-Complete, optimal solutions will require exponential time
- Can you come up with an algorithm that will give a (possiblly non-optimal) solution for the problem?

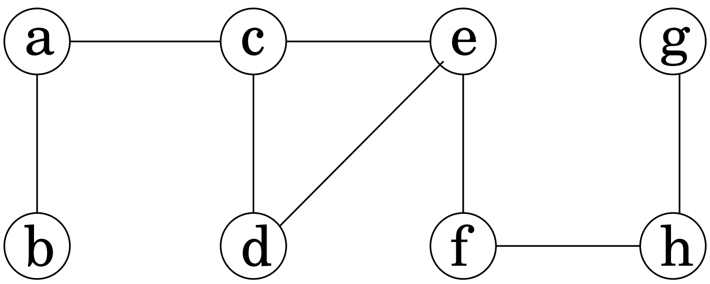
21-53: **Vertex Cover**

Approx-Vertex-Cover(V,E)
  $C \leftarrow \{\}$
  $E' \leftarrow E$
  while $E' \neq \{\}$
    let (u, v) be any edge in E'
    $C \leftarrow C \cup \{u, v\}$
    remove all edges from E' that contain
      u or v

21-54: **Vertex Cover**

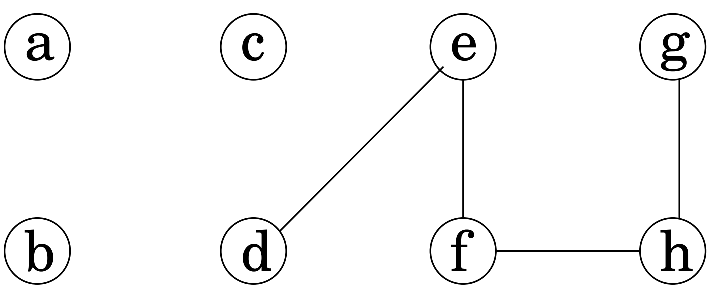


21-55: **Vertex Cover**



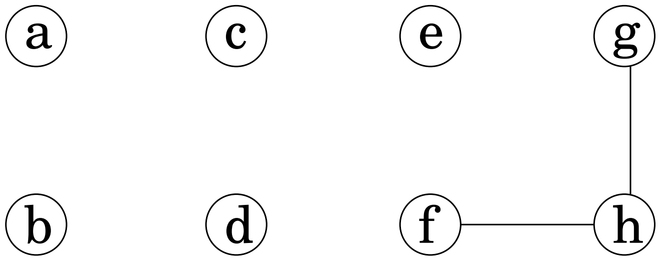
C = {a, c}

21-56: **Vertex Cover**



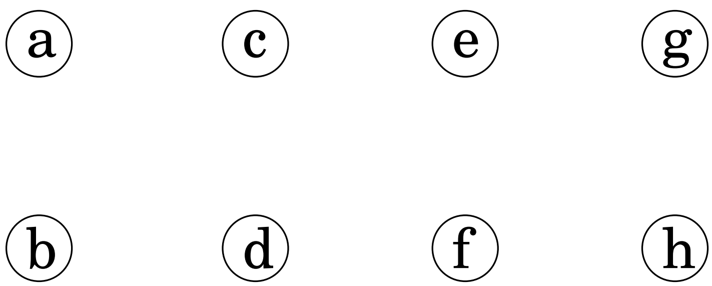
C = {a, c, d, e}

21-57: **Vertex Cover**



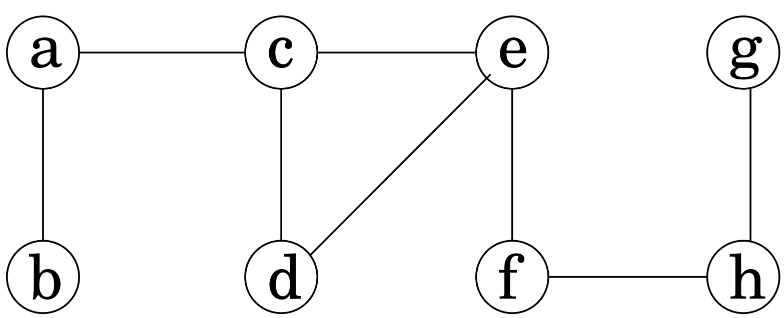
C = {a, c, d, e, f, h}

21-58: **Vertex Cover**



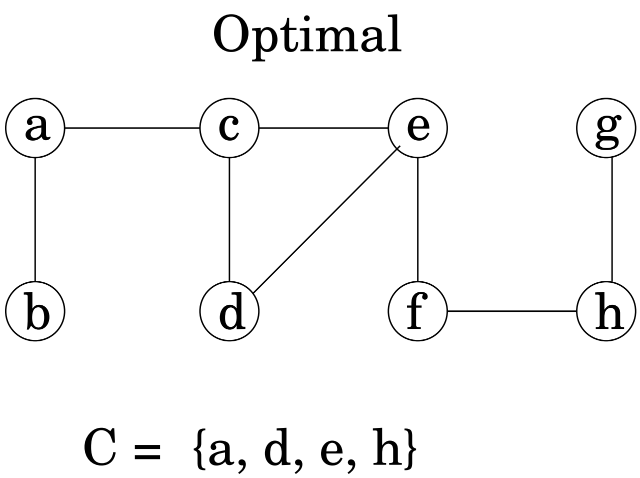
C = {a, c, d, e, f, h}

21-59: **Vertex Cover**



C = {a, c, d, e, f, h}

21-60: **Vertex Cover**



21-61: **Vertex Cover**

- Approx. Vertex-Cover is a polynomial-time 2-approximation algorithm
  - $\rho(n) = 2$
- Let C be the set of vertices found by approx. algorithm
- Let C^* be the optimal set of vertices
- $|C| \leq 2 * |C^*|$

21-62: **Vertex Cover**

- Let A be the set of edges selected by Approx. Vertex Cover
- Optimal vertex cover must pick at least one of the vertices for each edge in A
  - $|C^*| \geq |A|$
- Approx. vertex cover picked *both* vertices for each edge in A:
  - $|C| = 2 * |A|$
- Putting pieces together: $|C^*| \geq |A| = |C|/2$, $|C| \leq 2 * |C^*|$

21-63: **TSP**

- Travelling Salesman problem
  - Complete, undirected graph $G = (V, E)$
  - Cost for each edge
  - Find a cycle that includes vertices, that minimizes total cost

21-64: **TSP w/ triangle inequality**

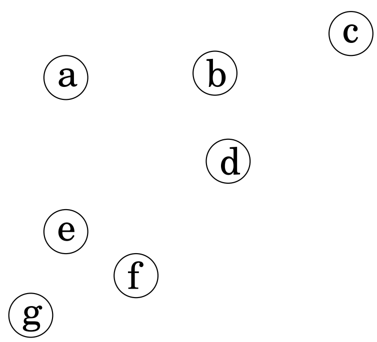
- TSP on plane
  - Each node has an x,y location
  - Cost bewteen nodes is the distance between nodes

- Slighly more general: TSP with triangle inequality
  - For any three vertices $v_1, v_2, v_3 \in V$, $c(v_1, v_2) + c(v_2, v_3) \geq c(v_1, v_3)$

21-65: **Approximate TSP**

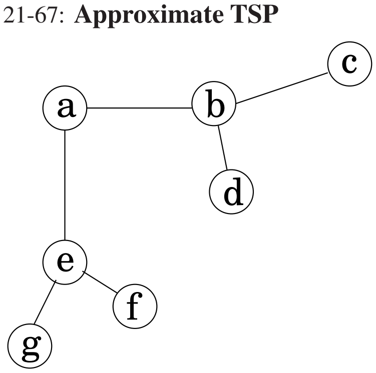
Approx-TSP(V, E, c)
  select any vertex $r \in V$ as root vertex
  Compute MST T of graph from root r using Prim
  $L \leftarrow$ list of vertices visited in preorder tree
    walk of T
  return L

21-66: **Approximate TSP**



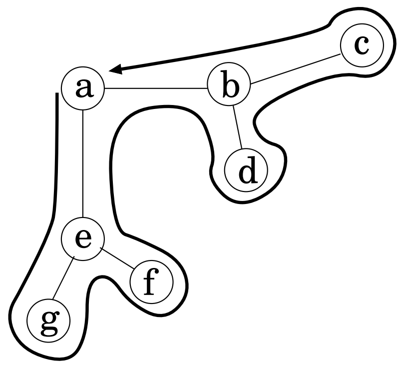
Edges between all pairs of vertices
cost = distance between vertices

21-67: **Approximate TSP**



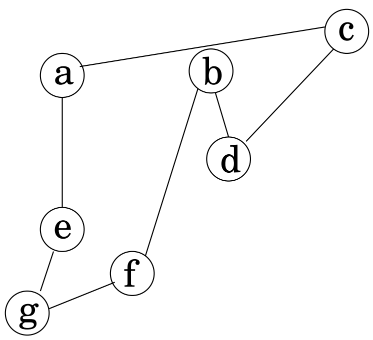
Start with vertex a
create MST

21-68: **Approximate TSP**



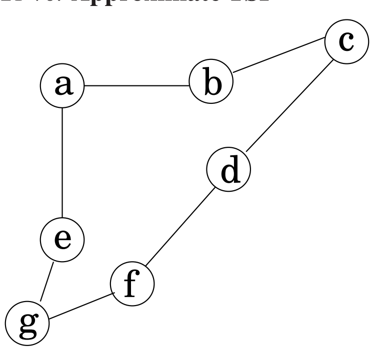
Preoder Traversal of MST
a,e,g,f,b,d,c

21-69: **Approximate TSP**



Traversal => Tour
a,e,g,f,b,d,c

21-70: **Approximate TSP**



Best TSP tour
a,e,g,f,d,c,b

21-71: **Approximate TSP**

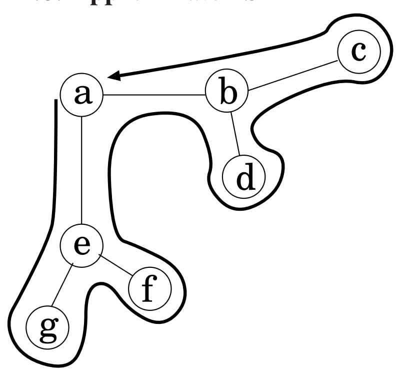
- Approximate-TSP finds a tour whose cost is at most twice the cost of the optimal TSP

- $\rho(n) \leq 2$
- Why?

21-72: **Approximate TSP**

- Cost of TSP Tour $\geq$ cost of MST
- Consider a "full walk" of MST (revisit vertices)

21-73: **Approximate TSP**



"full walk" of MST
a,e,g,e,f,e,a,b,d,b,c,b,a

21-74: **Approximate TSP**

- Cost of "full walk" = 2 * cost MST
  - Since we are following each edge twice
- Not a valid tour
  - Repeated vertices
- Remove repeated vertices, get preorder walk
  - Cost of preorder walk $\leq$ cost of full walk – triangle inequality

21-75: **Approximate TSP**

- Cost of approximate TSP tour $\leq$ cost of full walk
- Cost of full walk $\leq$ 2 * cost of MST
- Cost of MST $\leq$ cost of optimal TSP tour

Cost of approximate TSP tour $\leq$ 2 * cost of optimal tour

21-76: **General TSP**

- Alas, our algorithm does not generalize to all TSP
  - Relied on the triangle inequality
- No good approximate tours can be found in polynomial time for TSP, unless NP = P

- See text for proof

21-77: **Randomized Approximation**

- Randomized algorithms can be used to calculate approximate solutions
  - Unsurprising, we've used randomized algorithms to calculate exact values – Randomized Quicksort
- Randomized Approximation Algorithms are a little different
  - Random values that are picked affect the outcome
  - Instead of an approximation ratio, we have an *expected approximation ratio*

21-78: **Randomized MAX-3-SAT**

- MAX-3-SAT
  - Satisfiability Problem,
  - Each clause contains exactly 3 variables
  - No variable is repeated in the same clause
  - Trying to maximize the number of satisfied clauses

21-79: **Randomized MAX-3-SAT**

- Algorithm is extremely simple:
  - For each variable x_i:
    - Set $x_i =$True with Probability 0.5
- What is an upper limit to the expected approximation ratio?

21-80: **Randomized MAX-3-SAT**

- $Y_i = I\{$clause i is satisfied$\}$
  - So $Y_i =$true if at least one of the literals in the ith clause is set to 1
  - Setting of 3 literals in each clause is independent
  - $Pr\{$clause i is not satisfied$\}$ =

21-81: **Randomized MAX-3-SAT**

- $Y_i = I\{$clause i is satisfied$\}$
  - So $Y_i =$true if at least one of the literals in the ith clause is set to 1
  - Setting of 3 literals in each clause is independent
  - $Pr\{$clause i is not satisfied$\} = (1/2)^3 = 1/8$
  - $Pr\{$clause i is satisfied$\} = 1 - (1/2)^3 = 7/8$

21-82: **Randomized MAX-3-SAT**

- $Y_i = I\{$clause i is satisfied$\}$
- $Y =$number of satisfied clauses = $\sum_{i=1}^{m} Y_i$

- Assuming m clauses

$$
\begin{aligned}
E[Y] &= E[\sum_{i=1}^{m}] \\
&= \sum_{i=1}^{m} E[Y_i] \\
&= \sum_{i=1}^{m} 7/8 \\
&= 7m/8
\end{aligned}
$$

- Expected number of satisfied clauses: $7m/8$

21-83: **Randomized MAX-3-SAT**

- Finding the expected approximation ratio:
  - Largest possible number of satisfied clauses = m.
  - Expected number of satisfied clauses = $7m/8$
  - Maximum expected approximation ratio: $m/(7m/8) = 8/7$
- Pick values randomly, expected approximation ratio is at most $8/7$

21-84: **Subset-Sum Problem**

- Subset-Sum Decision Problem
- Given:
  - A set $S = \{x_1, x_2, x_3, \ldots x_n\}$ of positive integers
  - A target t
- Is there a subset of S that sums exactly to t?

21-85: **Subset-Sum Problem**

- Subset-Sum Optimization Problem
- Given:
  - A set $S = \{x_1, x_2, x_3, \ldots x_n\}$ of positive integers
  - A target t
- Find a subset of S with the largest possible sum less than or equal to t

21-86: **Subset-Sum Problem**

Exact-Subset-Sum(S, t)
  $n \leftarrow |S|$
  $L \leftarrow \{0\}$
  for $i \leftarrow 1$ to n
    $L \leftarrow$ MergeLists($L, L + S[i]$)
    Remove all elements larger than t from L
  return largest element in L

- $L + S[i]$ means add $S[i]$ to each element in L
- MergeLists: Merge two sorted lists, removing duplicates

21-87: **Subset-Sum Problem**

$S = \{1, 3, 5\}$

- $L = \{0\}$
- $L = \{0, 1\}$
- $L = \{0, 1, 3, 4\}$
- $L = \{0, 1, 3, 4, 5, 6, 8, 9\}$

21-88: **Subset-Sum Problem**

$S = \{1, 2, 3\}$

- $L = \{0\}$
- $L = \{0, 1\}$
- $L = \{0, 1, 2, 3\}$
- $L = \{0, 1, 2, 3, 4, 5, 6\}$

21-89: **Subset-Sum Problem**

- What is the worst-case running time?

21-90: **Subset-Sum Problem**

- What is the worst-case running time?
  - List L could be as large as 2^n
  - Running time is $O(2^n)$
  - (Polynomial if sum of all elements in L is bound by a polynomial in $|S|$)

21-91: **Subset-Sum Problem**

- Algorithm is exponential because L can grow exponentially large
- So, if we wanted an approximation in polynomial time, what could we do?

21-92: **Subset-Sum Problem**

- Algorithm is exponential because L can grow exponentially large

- So, if we wanted an approximation in polynomial time, what could we do?
  - Prune L to prevent it from getting too large
  - Removing the wrong element could prevent us from finding an optimal solution
  - How can we prune L to minimize / bound the error?

21-93: **Subset-Sum Problem**

- Basic idea:
  - After creating the list L, "trim" it by removing elements
  - If we have two elements that are close to each other, we remove the larger of them
    - Sum can be off by the difference of the elements

21-94: **Subset-Sum Problem**

- Function TRIM, takes as input a list and a δ, and trims all elements that are within δ % of the previous element in the list:

```
TRIM(L, δ)
  m ← |L|
  L' ← L[1]
  last ← L[1]
  for i ← 2 to m
    if L[i] >last ∗(1 + δ)
      append L[i] to L'
      last ← L[i]
  return L'
```

21-95: **Subset-Sum Problem**

```
Approx-Subset-Sum(S, t, ϵ)
  n ← |L|
  L ← {0}
  for i ← 1 to n
    L ← MergeLists(L, L + S[i])
    L ← TRIM(L, ϵ/2n)
    remove elements greater than t from L
  return largest element in L
```

- Returns an element within $(1 + \epsilon)$ of optimal

21-96: **Subset-Sum Problem**

$S = \{104, 102, 201, 101\}$, $t = 308$, $\epsilon = .4$, $\delta = 0.05$

- $L = \{0\}$
- $L = \{0, 104\}$
  - (no trimming)
- $L = \{0, 102, 104, 206\}$
  - $104 < 102 * 1.05$
- $L = \{0, 102, 206\}$

- $L = \{0, 102, 201, 206, 303, 407\}$
  - $206 < 201 * 1.05$
  - $407 > t$
- $L = \{0, 102, 201, 303\}$

21-97: **Subset-Sum Problem**

$S = \{104, 102, 201, 101\}$, $t = 308 \epsilon = .4$, $\delta = 0.05$

- $L = \{0, 102, 201, 303\}$
- $L = \{0, 101, 102, 201, 203, 302, 303, 404\}$
  - $102 < 101 * 1.05$
  - $203 < 201 * 1.05$
  - $303 < 302 * 1.05$
  - $404 > \epsilon$
- $L = \{0, 101, 201, 302\}$
- Result: 302
- Optimal: 307 (104 + 102 + 101)
- Within 0.40 of optimal

21-98: **Subset-Sum Problem**

- Approx-Subset-Sum(S, t, ϵ)
  - Always returns a result within $(1 + \epsilon)$ of the true optimal
  - Runs in time polynomial in length of input and $1/\epsilon$

21-99: **Subset-Sum Problem**

- Runs in time polynomial in length of input and $1/\epsilon$:
  - First, we'll find a bound on how long each list L_i can be
  - After each trimming, consider successive elements z, z'
  - $z'/z > 1 + \epsilon/2n$
  - Largest that L_i could be:
    - $0, 1, \epsilon/2n, 2\epsilon/2n, 3\epsilon/2n \ldots$
  - size of $L_i < \log_{1+\epsilon/2n} t + 2$

21-100: **Subset-Sum Problem**

- size of $L_i < \log_{1+\epsilon/2n} t$

$$\begin{aligned}
\log_{1+\epsilon/2n} t &= \frac{\ln t}{\ln(1 + \epsilon/2n)} + 2 \\
&\leq \frac{2n(1 + \epsilon/2n) \ln t}{\epsilon} + 2 \\
&\leq \frac{4n \ln t}{\epsilon} + 2
\end{aligned}$$

- Bound is clearly polynnomial in size of input and $\frac{1}{\epsilon}$

$\frac{x}{1+x} \leq \ln(1 + x) \leq x$, $0 < \epsilon < 1$

21-101: **Subset-Sum Problem**

- Always returns a result within $(1 + \epsilon)$ of the true optimal
  - See text, pg. 1048