Complexity

Nishant Mehta

Lectures 17–19

“Easy” Problems vs “Hard” Problems

- **Minimum Spanning Tree problem**
- Input: A weighted, undirected graph $G = (V, E)$
- Output: A spanning tree $T \subseteq E$ of minimum cost

- How hard? For n vertices, there are n^{n-2} spanning trees (exponentially many!)

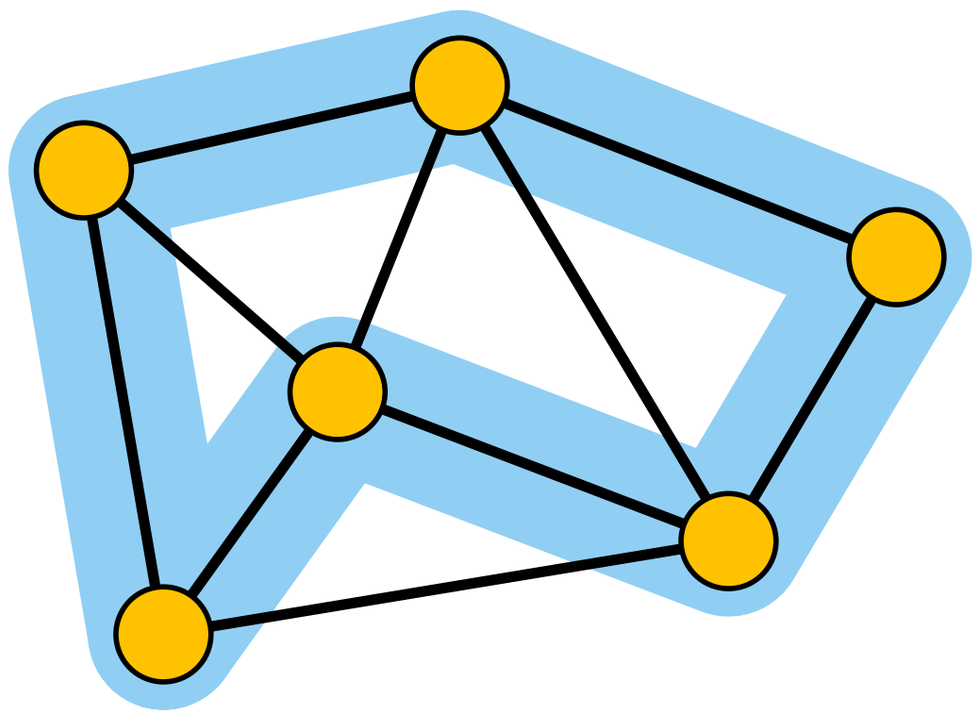
- Yet, we have fast algorithms for the MST Problem.
  - Prim’s algorithm: $O(m \log n)$ for n vertices and m edges

“Easy” Problems vs “Hard” Problems

- **Traveling Salesperson Problem (TSP)**
- Input: A complete, weighted, undirected graph $G = (V, E)$
- Output: A *tour* $C \subseteq E$ of minimum cost

also called *Hamiltonian cycle*

- A ***tour***: A cycle that visits each vertex exactly once.



“Easy” Problems vs “Hard” Problems

- **Traveling Salesperson Problem (TSP)**
- Input: A complete, weighted, undirected graph $G = (V, E)$
- Output: A *tour* $C \subseteq E$ of minimum cost

- There are $\frac{(n-1)!}{2}$ tours, so exhaustive search is intractable
- No efficient algorithms are known for TSP!
- That is, no polynomial-time algorithm is known for TSP, so it is not known if there exists a constant c such that there is an algorithm that can solve TSP in time $O(n^c)$

I conjecture that there is no good algorithm for the traveling saleman problem. My reasons are the same as for any mathematical conjecture: (1) It is a legitimate mathematical possibility, and (2) I do not know.

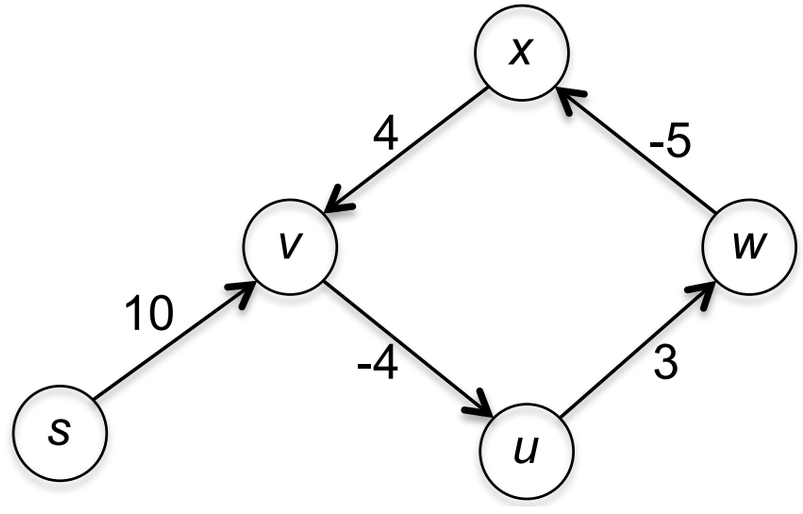
Jack Edmonds, photo from 2020

quote from 1967

“Easy” Problems vs “Hard” Problems

- **Single-Source Shortest Path problem**
- Input: A weighted, directed graph $G = (V, E)$, a source vertex $s \in V$, and a destination vertex $t \in V$
- Output: An s-t path of minimum weight or “False” if no shortest s-t path exists

- Remember: if there exists an s-t path that contains a negative cycle, then no shortest s-t path exists



“Easy” Problems vs “Hard” Problems

- **Single-Source Shortest Path problem**
- Input: A weighted, directed graph $G = (V, E)$, a source vertex $s \in V$, and a destination vertex $t \in V$
- Output: An s-t path of minimum weight or “False” if no shortest s-t path exists

- How hard? There can be exponentially many paths
- Yet again, we have fast algorithms
  - Bellman-Ford: $O(mn)$ for n vertices and m edges

“Easy” Problems vs “Hard” Problems

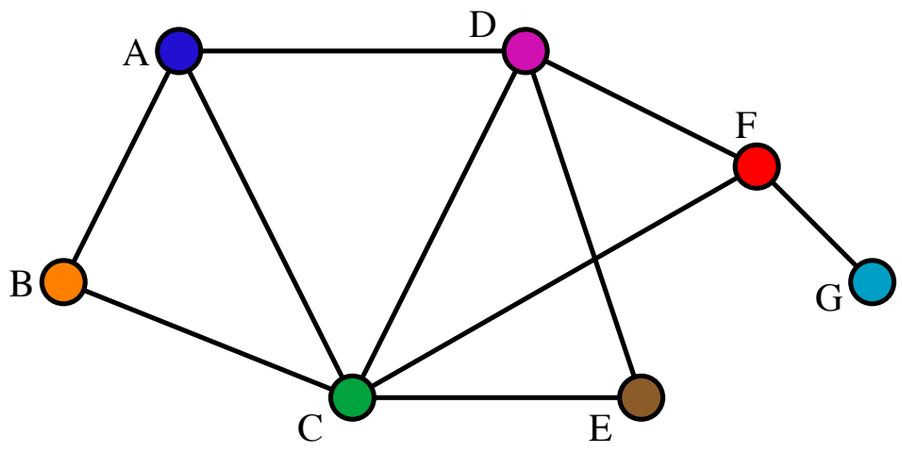
- **Cycle-Free Shortest Path problem**
- Input: A weighted, directed graph $G = (V, E)$, a source vertex $s \in V$, and a destination vertex $t \in V$
- Output: An cycle-free s-t path of minimum weight

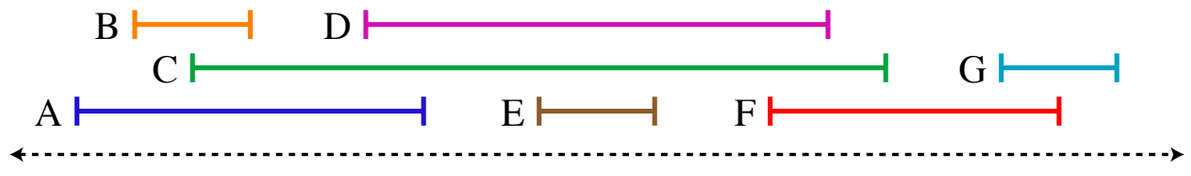
- No efficient algorithms are known for Cycle-Free Shortest Path!

“Easy” Problems vs “Hard” Problems

- **Weighted Independent Set for Interval Graphs**
- Input: An undirected graph $G = (V, E)$ that is an *interval graph*, where each vertex $v \in V$ is associated with a weight $w_v \in \mathbb{R}$
- Output: An *independent set* $S \subseteq V$ of maximum weight

interval graph





“Easy” Problems vs “Hard” Problems

- **Weighted Independent Set for Interval Graphs**
- Input: An undirected graph $G = (V, E)$ that is an *interval graph*, where each vertex $v \in V$ is associated with a weight $w_v \in \mathbb{R}$
- Output: An *independent set* $S \subseteq V$ of maximum weight

- Does this sound familiar?
  It is the same as Weighted Interval Scheduling!

- We saw a dynamic programming-based algorithm that has runtime $O(n \log n)$

“Easy” Problems vs “Hard” Problems

- **Weighted Independent Set (for General Graphs)**
- Input: An undirected graph $G = (V, E)$, where each vertex $v \in V$ is associated with a weight $w_v \in \mathbb{R}$
- Output: An *independent set* $S \subseteq V$ of maximum weight

- No efficient algorithms are known for Weighted Independent Set

Polynomial Time Algorithms

- MergeSort for Sorting
- Kruskal's Algorithm for MST
- Dijkstra's Algorithm for Single Source Shortest Paths
- Floyd-Warshall for All-Pairs Shortest Paths
- "Unnamed DP Algorithm" for Weighted Interval Scheduling

Problems that can be solved in Polynomial Time

- Sorting
- MST
- Single Source Shortest Paths
- All-Pairs Shortest Paths
- Weighted Interval Scheduling

Decision, Search, and Optimization problems

- Decision problem: Output “yes” if a feasible solution exists and “no” otherwise
  - Example: Directed Hamiltonian Cycle

Decision, Search, and Optimization problems

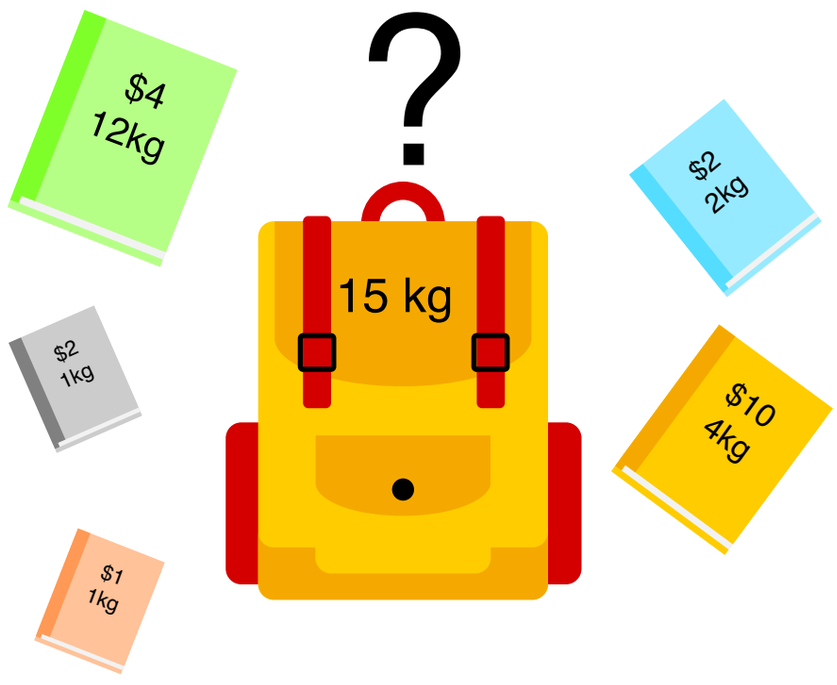
- Decision problem: Output "yes" if a feasible solution exists and "no" otherwise
  - Example: Directed Hamiltonian Cycle
- Search problem: Output a feasible solution if one exists and "False" (or "no solution") otherwise
  - Example: Search problem of TSP (does there exist a solution of cost at most k?)

Decision, Search, and Optimization problems

- Decision problem: Output "yes" if a feasible solution exists and "no" otherwise
  - Example: Decision version of TSP (does there exist a solution of cost at most k?)
- Search problem: Output a feasible solution if one exists and "False" (or "no solution") otherwise
  - Example: Search version of TSP (find a solution of cost at most k)
- Optimization problem: Output a feasible solution of minimum cost (for minimization) or maximum value (for maximization) or "False" if no feasible solution exists
  - Example: TSP (find a minimum cost solution)

“Hard” Problems: Knapsack

- **Knapsack**
- Input: n item values $v_1, \ldots, v_n$, n item weights $w_1, \ldots, w_n$, and a weight capacity C
- Output: Among all subsets $S \subseteq [n]$ satisfying the weight constraint $\sum_{i \in S} w_i \leq C$, find a subset of maximum value $\sum_{i \in S} v_i$



“Hard” Problems: Knapsack

- **Knapsack**
- Input: n item values $v_1, \ldots, v_n$, n item weights $w_1, \ldots, w_n$, and a weight capacity C
- Output: Among all subsets $S \subseteq [n]$ satisfying the weight constraint $\sum_{i \in S} w_i \leq C$, find a subset of maximum value $\sum_{i \in S} v_i$

- In an earlier lecture, we saw that dynamic programming can be used to get an algorithm that runs in time $O(n\,C)$
- Recall that this runtime is not polynomial in input size. Why? The space needed to store C is only $\log_2 C$.

“Hard” Problems: Knapsack

- **Knapsack**
- Input: n item values $v_1, \ldots, v_n$, n item weights $w_1, \ldots, w_n$, and a weight capacity C
- Output: Among all subsets $S \subseteq [n]$ satisfying the weight constraint $\sum_{i \in S} w_i \leq C$, find a subset of maximum value $\sum_{i \in S} v_i$

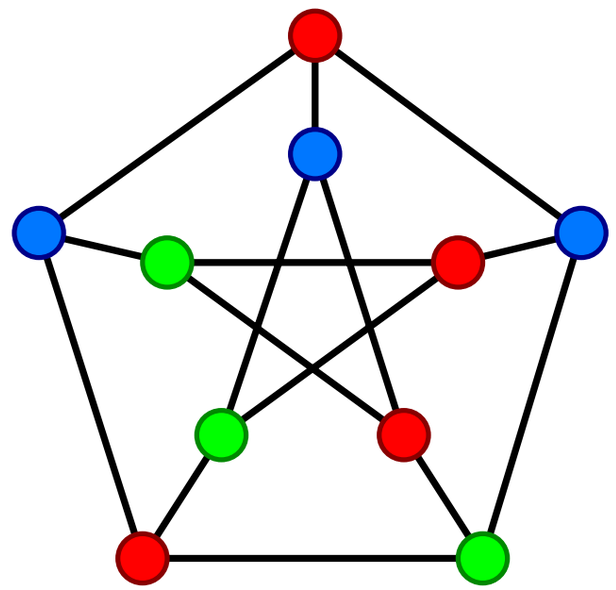
- No efficient algorithm is known for Knapsack

"Hard" Problems: k-coloring

- k-**Coloring**, for parameter k
- Input: An undirected graph $G = (V, E)$
- Output: A k-coloring of G, or "False" if G is not k-colorable

- A *k-coloring* is an assignment of a color from $\{1,2,\ldots,k\}$ to each vertex so that no adjacent vertices share a color
- A graph is *k-colorable* if it has a k-coloring

a 3-coloring



"Hard" Problems: k-coloring

- k-**Coloring**, for parameter k
- Input: An undirected graph $G = (V, E)$
- Output: A 2-coloring of G, or "False" if G is not 2-colorable

- 2-coloring can be solved in polynomial time: it can be solved in time $O(n + m)$ using either DFS or BFS

- No efficient algorithm is known for 3-coloring

“Hard” Problems - Satisfiability

- **2-SAT**
- Input: A boolean expression over n variables, expressed as a formula in *2-CNF*
- Output: A satisfying assignment if one exists or “False” if no such assignment exists

- “CNF” stands for *conjunctive normal form*. A boolean expression is in ***k-CNF*** if it is a conjunction of clauses, where each clause is a disjunction of at most k *literals*
- Example of 2-CNF: $(x_2 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_3) \wedge (x_5) \wedge (x_3 \vee \bar{x}_5)$

literals ($\bar{x}_4$ is the negation of x_4)

“Hard” Problems - Satisfiability

- **2-SAT**
- Input: A boolean expression over n variables, expressed as a formula in *2-CNF*
- Output: A satisfying assignment if one exists or “False” if no such assignment exists

- 2-SAT can be solved in polynomial time (i.e., there is an algorithm that solves 2-SAT in polynomial time)

“Hard” Problems - Satisfiability

- **3-SAT**
- Input: A boolean expression over n variables, expressed as a formula in 3-*CNF*
- Output: A satisfying assignment if one exists or “False” if no such assignment exists

- Example of 3-SAT:
  - $(x_2 \vee \bar{x}_4 \vee x_5) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \wedge (x_1 \vee x_5 \vee \bar{x}_6) \wedge (x_3 \vee \bar{x}_2)$
- No efficient algorithm is known for 3-SAT

“Hard” Problems

- We have seen various problems that seem to be hard:
  - Traveling Salesperson Problem
  - Cycle-Free Shortest Path
  - Weighted Independent Set
  - Knapsack
  - 3-Coloring
  - 3-SAT
- And there are many other interesting problems for which really smart people have failed to devise efficient algorithms
- So far, our notion of hardness seems a bit heuristic. How can we argue more formally that a given problem is hard?

From “Hard” to Hard: formalizing hardness

- How to formalize what it means for a problem to be hard?

- **Idea 1**: Prove that no polynomial-time algorithm exists.
  How? Devise a worst-case lower bound on the runtime of *any* algorithm for the problem (analogous to the lower bound for comparison-based sorting algorithms)

- Unfortunately, for all the “hard” problems considered so far (including many that we didn’t cover so far!), no such lower bounds are known

- **Idea 2**: Consider *relative hardness* instead of *absolute hardness*

Relative Hardness

- TSP seems hard. Why? Because a lot of people spent a lot of time trying to find polynomial-time algorithms for it

- But, if someone found a polynomial-time algorithm for TSP, the only implication for our view of the world (our theory) is that TSP can be solved in polynomial time. What if the implications were far greater?

- What if a polynomial time algorithm for TSP implied the existence of polynomial time algorithms for a LARGE collection of problems, none of which we previously could solve in polynomial time (despite lots of effort spent on each of these problems)?!

The Complexity Class NP

- Somewhat informally, a problem is in the complexity class **NP** if both:
  - any candidate solution can be expressed using a description length (number of bits) at most polynomial in the input size
    ("so, we can read solutions efficiently")
  - any candidate solution to the problem, the feasibility of the solution can be verified in polynomial time
    ("so, we can efficiently verify whether an alleged solution does in fact solve the problem")

The Complexity Class NP (informal)

- Even more informally, a problem is in $\mathbf{NP}$ if, whenever someone offers a solution that they claim solves the problem, we can efficiently verify whether their solution indeed solves the problem
- Even more succinctly, "a problem is in $\mathbf{NP}$ if verification is cheap"

The Complexity Class NP (informal)

- Even more informally, a problem is in $\mathbf{NP}$ if, whenever someone offers a solution that they claim solves the problem, we can efficiently verify whether their solution indeed solves the problem
- Even more succinctly, "a problem is in $\mathbf{NP}$ if verification is cheap"
- Example:
  - Consider the Search version of TSP: find a solution to TSP whose cost is at most k
  - Any candidate solution (a tour) is a sequence of n edges. So, it can be encoded using space linear in the size of the input
  - Also, we can compute the cost of any candidate solution by summing the weights of the edges (which is again linear in the size of the input). Finally, we compare the cost to k

NP-Hardness

- Informally, a problem is NP-Hard if an efficient algorithm for the problem implies efficient algorithms for *all* problems for which verifying a solution is easy

- Somewhat more formally, a problem is NP-Hard if an efficient algorithm for the problem implies efficient algorithms FOR ALL PROBLEMS in $\mathbf{NP}$

The Complexity Class P

- We say that a search problem belongs to the complexity class $\mathbf{P}$ if and only if the problem belongs to $\mathbf{NP}$ (so, solutions can be verified in polynomial time) and can be solved in polynomial time

- So, by definition, we have $\mathbf{P} \subseteq \mathbf{NP}$

- That is, any problem in $\mathbf{P}$ is also in $\mathbf{NP}$

- This direction passes a basic sanity check:
  - *Verifying* a solution should not be harder than *finding* a solution

P ≠ NP conjecture (informal)

- What about the opposite direction? Is every problem in $\mathbf{NP}$ also in $\mathbf{P}$?

- The P ≠ NP Conjecture:
  - There exists a problem for which solutions can be verified in polynomial time but which cannot be solved in polynomial time.

What does it all mean?

- Suppose a problem is NP-hard and someone finds a polynomial time algorithm for the problem. Then for all problems in $\mathbf{NP}$ (all problems for which it is "easy" to verify a solution), it is also "easy" to **find** a solution. That is:

  - For all problems where we can verify a solution in polynomial time, finding a solution is as easy as verifying a solution!

  - In particular, it would be true that $\mathbf{P} = \mathbf{NP}$

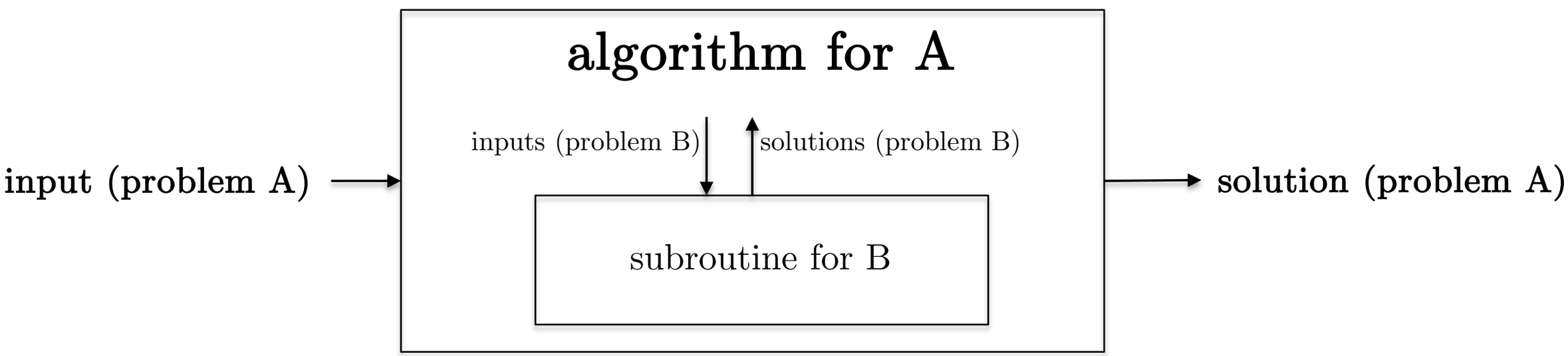
What if P ≠ NP?

- If $\mathbf{P} \neq \mathbf{NP}$, does that mean problems that are in $\mathbf{NP}$ but not in $\mathbf{P}$ require exponential time to solve?
  - Not necessarily. There are things in between polynomial time and exponential time, like $2^{\sqrt{n}}$
    - Also, experts believe that the problem of Factoring cannot be done in polynomial time, but subexponential-time algorithms are known for Factoring

Reductions (Cook Reductions)

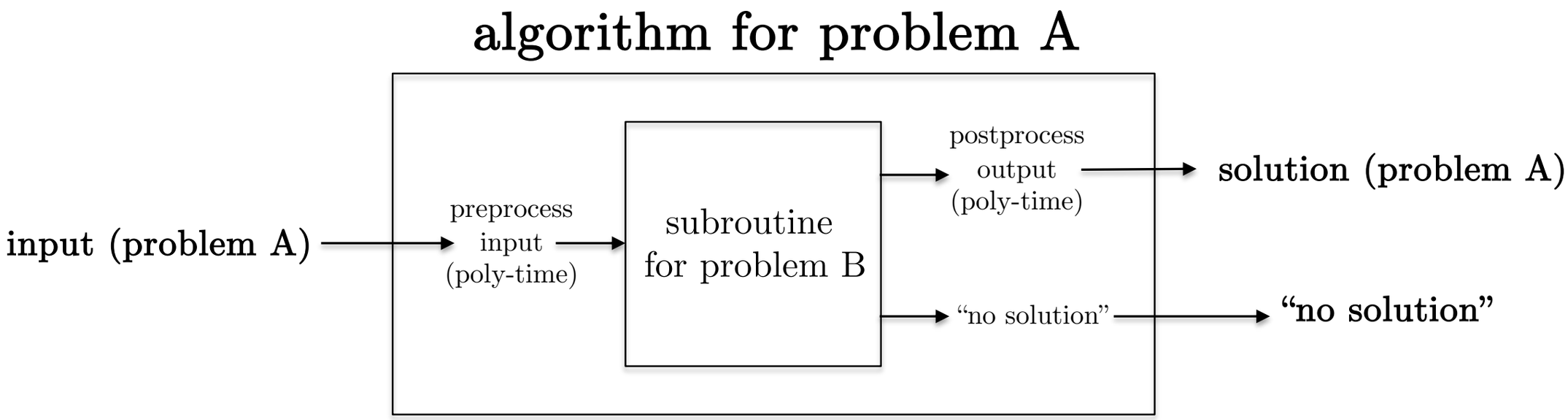
- We say that a problem A reduces (or can be reduced to) another problem B if problem A can be solved using a polynomial number of calls to an algorithm that solves problem B ("a subroutine for B") plus a polynomial amount of additional work

- Typical examples of this additional work:
  - The work for forming inputs to the subroutine for B
  - The work for translating the solution(s) for B to get a solution for A



Reductions (Levin Reductions)

- A Levin reduction from problem A to problem B works follows:
  - Given an instance of problem A, transform it (in polynomial time) to an instance for problem B
  - Pass this instance for problem B into the subroutine for B
  - If the subroutine for B returns a solution, transform the solution (in polynomial time) to a solution to problem A. If the subroutine for B returns "no solution", then return "no solution"

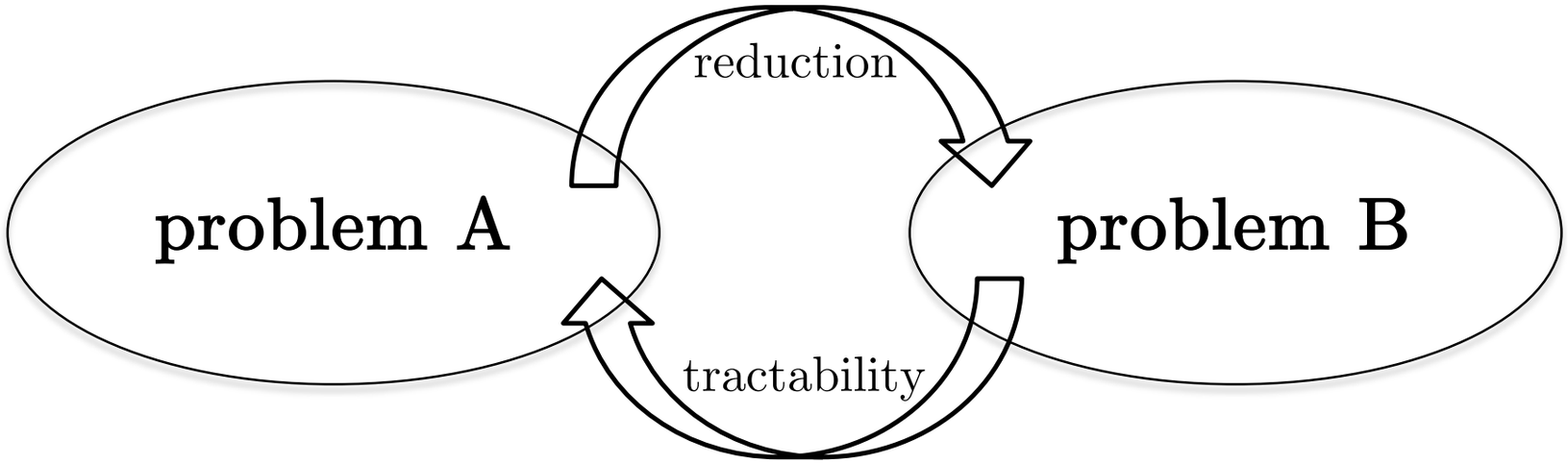


Reductions that we have already seen

- Super-basic examples:
  - All-Pairs Shortest Path reduces to Single-Source Shortest Path
  - Finding the median reduces to selecting the k^{th} smallest number
- Somewhat more involved examples:
  - Shortest Path reduces to Longest Path
  - Finding a bipartite matching reduces to finding a maximum flow

Reductions spread tractability in the reverse direction

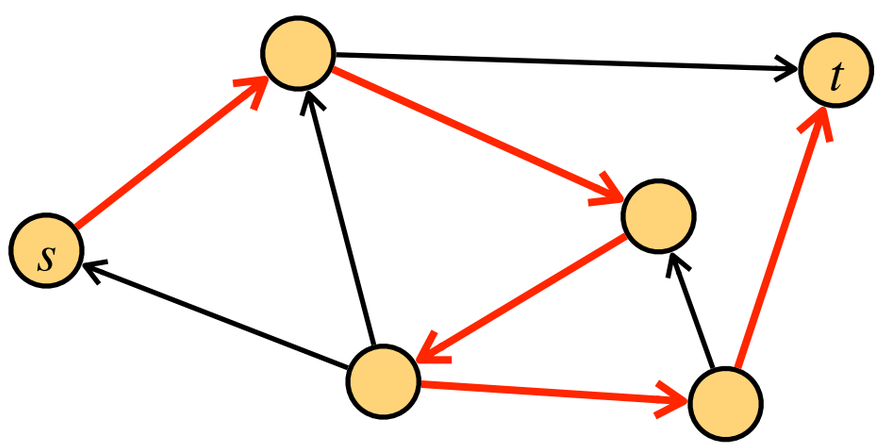
- Suppose Problem A reduces to Problem B and Problem B can be solved in polynomial time
- Then Problem A can be solved in polynomial time



Example: DHP reduces to Cycle-Free Shortest Path

- **Directed Hamiltonian Path (DHP)**
- Input: A directed graph $G = (V, E)$, a source vertex $s \in V$, and a destination vertex $t \in V$
- Output: An *s-t Hamiltonian path* if one exists, or "False" if no such path exists

- A ***Hamiltonian path*** is a path that visits each vertex exactly once



Example: DHP reduces to Cycle-Free Shortest Path

- **Directed Hamiltonian Path (DHP)**
- Input: A directed graph $G = (V, E)$, a source vertex $s \in V$, and a destination vertex $t \in V$
- Output: An *s-t Hamiltonian path* if one exists, or "False" if no such path exists

- **Claim**: DHP reduces to Cycle-Free Shortest Path

Example: DHP reduces to Cycle-Free Shortest Path

- **Directed Hamiltonian Path (DHP)**
- Input: A directed graph $G = (V, E)$, a source vertex $s \in V$, and a destination vertex $t \in V$
- Output: An *s-t Hamiltonian path* if one exists, or "False" if no such path exists

- **Claim:** DHP reduces to Cycle-Free Shortest Path

Note: We don't know any efficient algorithms for Cycle-Free Shortest Path. The claim is still useful though. If someone found an efficient algorithm for Cycle-Free Shortest Path, then the claim implies that we also have an efficient algorithm for DHP.

Reducing DHP to Cycle-Free Shortest Path

- Step 1: Transform DHP problem instance to a Cycle-Free Shortest Path problem instance
  - Cycle-Free Shortest Path needs a weighted, directed graph, a source vertex, and a destination vertex
  - Produce such a graph by setting all edge weights to -1 and use the same s and t as before
- Step 2: Feed constructed instance to algorithm for Cycle-Free Shortest Path
- Step 3: Recover solution for DHP from solution to Cycle-Free Shortest Path
  - Consider shortest path that was found. If cost of this path is $-(n-1)$, return the path that was found. Otherwise, return "no solution"

Reducing DHP to Cycle-Free Shortest Path

- Step 1: Transform DHP problem instance to a Cycle-Free Shortest Path problem instance
  - Cycle-Free Shortest Path needs a weighted, directed graph, a source vertex, and a destination vertex
  - Produce such a graph by setting all edge weights to -1 and use the same s and t as before

- Step 2: Feed constructed instance to algorithm for Cycle-Free Shortest Path

- Step 3: Recover solution for DHP from solution to Cycle-Free Shortest Path
  - Consider shortest path that was found. If cost of this path is $-(n-1)$, return the path that was found. Otherwise, return "no solution"

Note: Here, we are using the Search version of Cycle-Free Shortest Path: return a cycle-free shortest path of cost at most $-(n-1)$ if one exists, and return "no solution" otherwise

Toward a formal definition of NP-Hardness

- With the idea of Cook reductions, we are ready to formally define what is means for a problem to be NP-Hard

- But first, let's review our "informal" definition of the complexity class $\mathbf{NP}$ (which was actually formal enough for this lecture)

Recall: The Complexity Class NP

- Somewhat informally, a problem is in the complexity class **NP** if both:
  - any candidate solution can be expressed using a description length (number of bits) at most polynomial in the input size
    ("so, we can read solutions efficiently")
  - any candidate solution to the problem, the feasibility of the solution can be verified in polynomial time
    ("so, we can efficiently verify whether an alleged solution does in fact solve the problem")

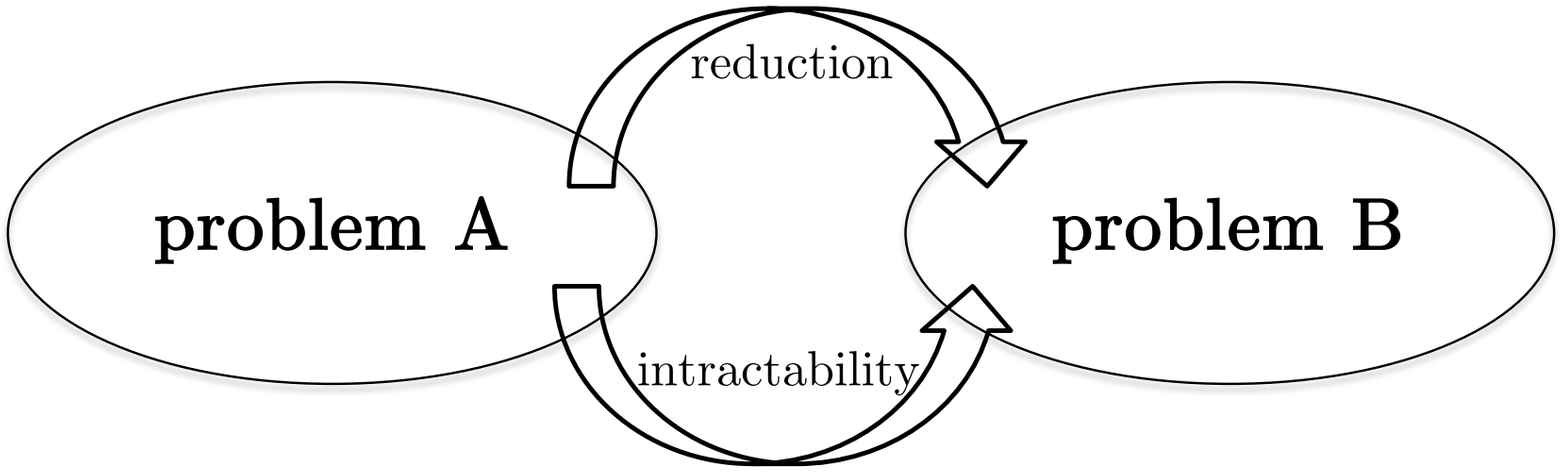
Formal Definition of NP-Hardness

- Recall our somewhat informal definition of NP-Hardness from earlier:
  - "a problem is NP-Hard if an efficient algorithm for the problem implies efficient algorithms FOR ALL PROBLEMS in $\mathbf{NP}$"

- Now that we have formally defined Cook reductions and the complexity class $\mathbf{NP}$, we can formally define what is means for a problem to be NP-Hard:
  - Formally, a problem is NP-Hard if every problem in $\mathbf{NP}$ reduces to it.

Reductions spread hardness in the forward direction

- Suppose problem A reduces to problem B and problem A is NP-Hard.
- Then problem B is also NP-Hard



How to prove that a problem is NP-Hard

- How can we prove that a problem B is NP-Hard?
- 2-step recipe:
    1) Identify a problem A that is NP-Hard
    2) Reduce problem A to problem B
- Sanity check:
    - Reductions spread intractability in the forward reduction
    - If we can efficiently solve problem B, then we can efficiently solve problem A; hence, since A is NP-Hard and we can (via the reduction) efficiently solve problem A, we can efficiently solve all problems in $\mathbf{NP}$.

How to prove that a problem is NP-Hard

- How can we prove that a problem B is NP-Hard?
- 2-step recipe:
  1) Identify a problem A that is NP-Hard
  2) Reduce problem A to problem B

Wait… so to prove that a problem is NP-Hard, we need to start with another problem that is NP-Hard?

Do NP-Hard problems even exist?!

- Sanity check:
  - Reductions spread intractability in the forward reduction
  - If we can efficiently solve problem B, then we can efficiently solve problem A; hence, since A is NP-Hard and we can (via the reduction) efficiently solve problem A, we can efficiently solve all problems in $\mathbf{NP}$.

How to Bootstrap: Cook-Levin Theorem

- NP-Hard problems do exist!
- **Cook-Levin Theorem:** 3-SAT is NP-Hard
- Consider the implications of this result:
  - Recall that a problem B is NP-Hard if any problem in $\mathbf{NP}$ reduces to B
  - So, any problem with polynomial-time-verifiable solutions can be reduced to 3-SAT!
- 3-SAT was the first problem ever proved to be NP-Hard, so:
  - the proof had to proceed "from scratch", by directly showing that if a problem has candidate solutions that can be verified in polynomial time, then the problem reduces to 3-SAT

Lots of problems are NP-Hard

- Loads of interesting problems are NP-Hard:
  - 3-SAT
  - SAT. Why? Trivially, 3-SAT reduces to it
  - 3-colorability. Why? Theorem: 3-SAT reduces to it
  - Directed Hamiltonian Path (DHP). Why? 3-SAT reduces to it
  - Cycle-Free Shortest Path. Why? As we saw, DHP reduces to it
  - Undirected Hamiltonian Path (UHP). Why? DHP reduces to it
  - Traveling Salesperson Problem. Why? UHP reduces to it

Lots of problems are NP-Hard

- And more NP-Hard problems:
  - Independent Set (IS). Why? 3-SAT reduces to it
  - Weighted Independent Set. Why? IS reduces to it
  - Clique. Why? IS reduces to it
  - Subset Sum. Why? IS reduces to it
  - Knapsack. Why? Subset Sum reduces to it

Showing reductions can be difficult

- It frequently is the case that reducing a known NP-Hard problem to another problem is not easy.
    - This includes some of the reductions referred to on the previous two slides
- It's useful to see how some of these trickier reductions proceed to:
    - gain experience in showing reductions yourself
    - convince you that problems that have been claimed to be NP-Hard actually are NP-Hard

Independent Set is NP-Hard

- We will consider just one of the more tricky reductions

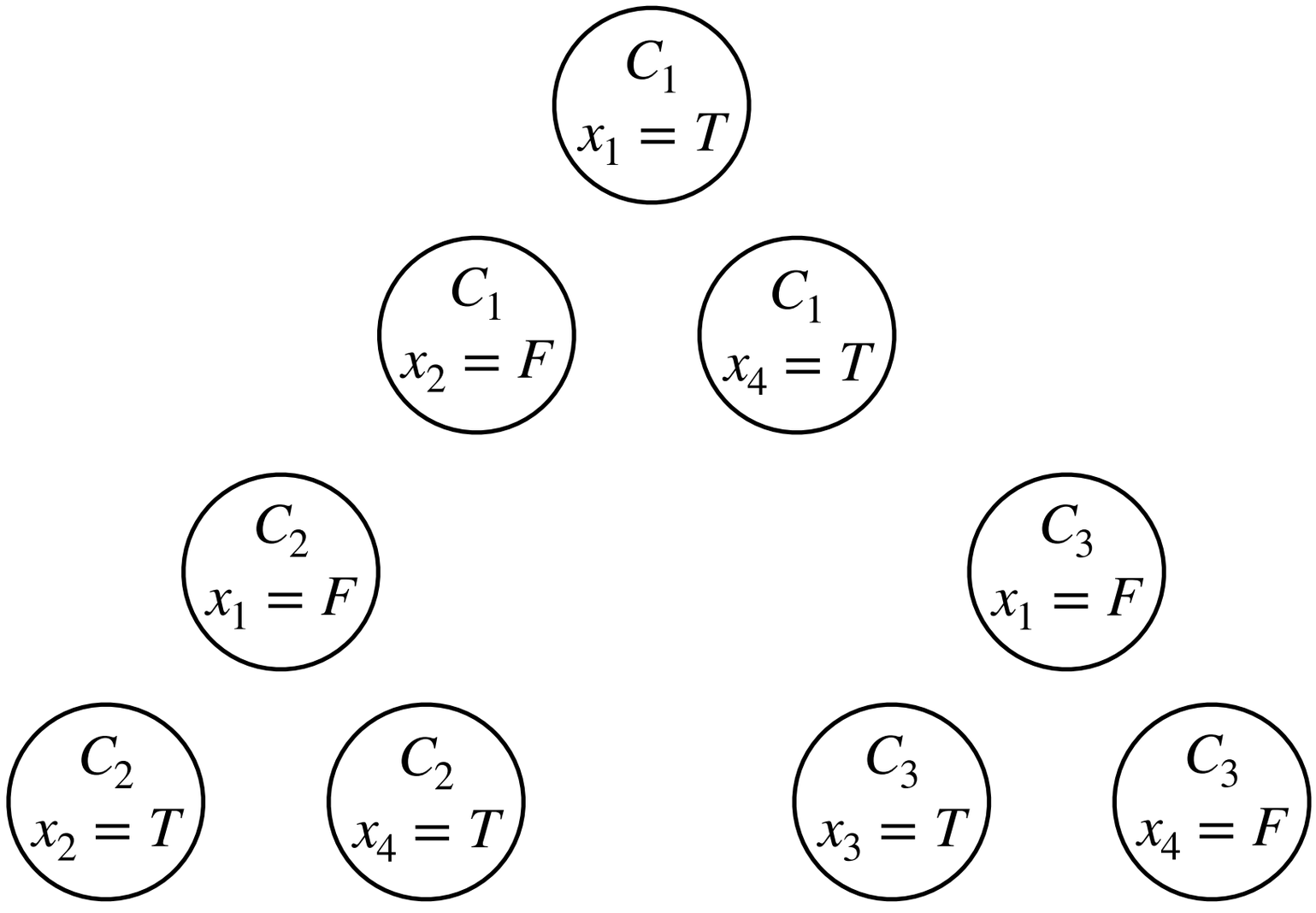
- **Independent Set** (the Search version)
- Input: An undirected graph $G = (V, E)$ and integer k
- Output: An *independent set* $S \subseteq V$ of size at least k if one exists and "no solution" otherwise

- **Theorem:** 3-SAT reduces to Independent Set
- Corollary: Independent Set is NP-Hard
    - Proof of the corollary: 3-SAT is NP-Hard and Independent Set reduces to 3-SAT

3-SAT Reduces to Independent Set

Idea: Given a 3-SAT instance, create vertex for each literal of each clause

$$\underset{C_1}{(x_1 \vee \bar{x}_2 \vee x_4)} \wedge \underset{C_2}{(\bar{x}_1 \vee x_2 \vee x_4)} \wedge \underset{C_3}{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}$$

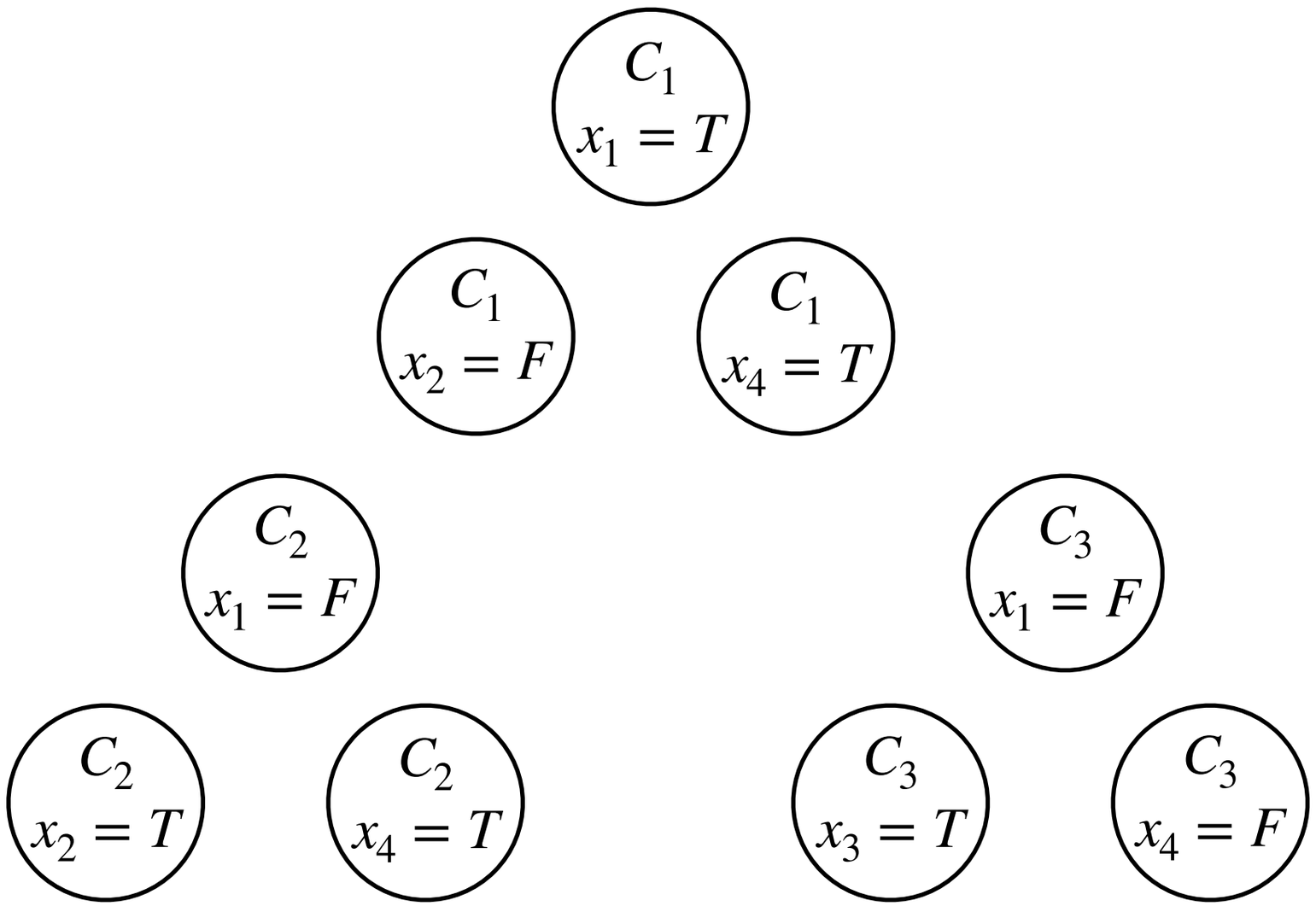


3-SAT Reduces to Independent Set

Idea: Given a 3-SAT instance, create vertex for each literal of each clause

$$\underbrace{(x_1 \vee \bar{x}_2 \vee x_4)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}_{C_3}$$

Let's make vertices in an independent set correspond to an assignment to the variables



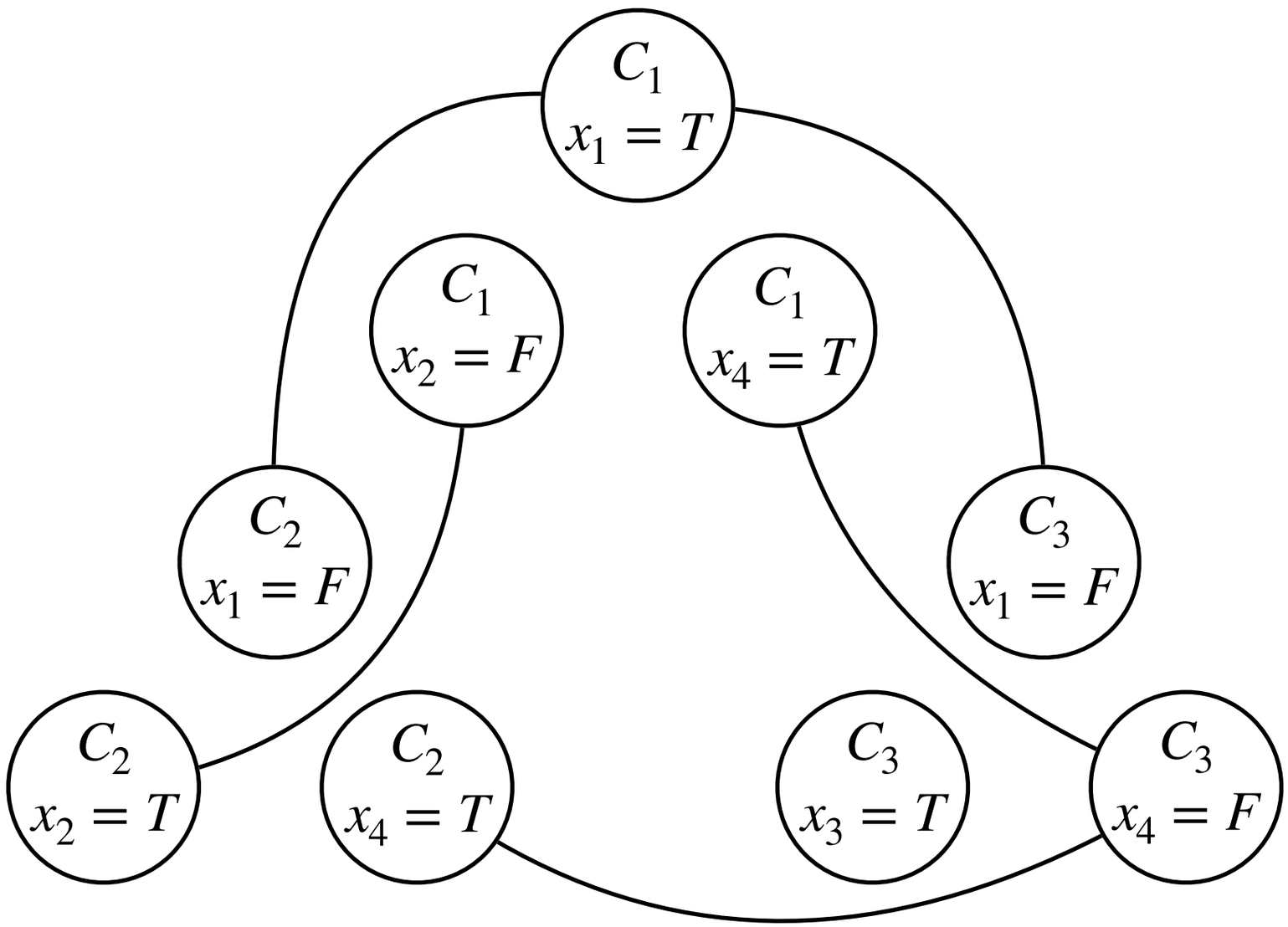
3-SAT Reduces to Independent Set

Idea: Given a 3-SAT instance, create vertex for each literal of each clause

$$\underbrace{(x_1 \vee \bar{x}_2 \vee x_4)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}_{C_3}$$

Let's make vertices in an independent set correspond to an assignment to the variables

Some vertices (variable assignments) are in conflict, so add edges between them



3-SAT Reduces to Independent Set

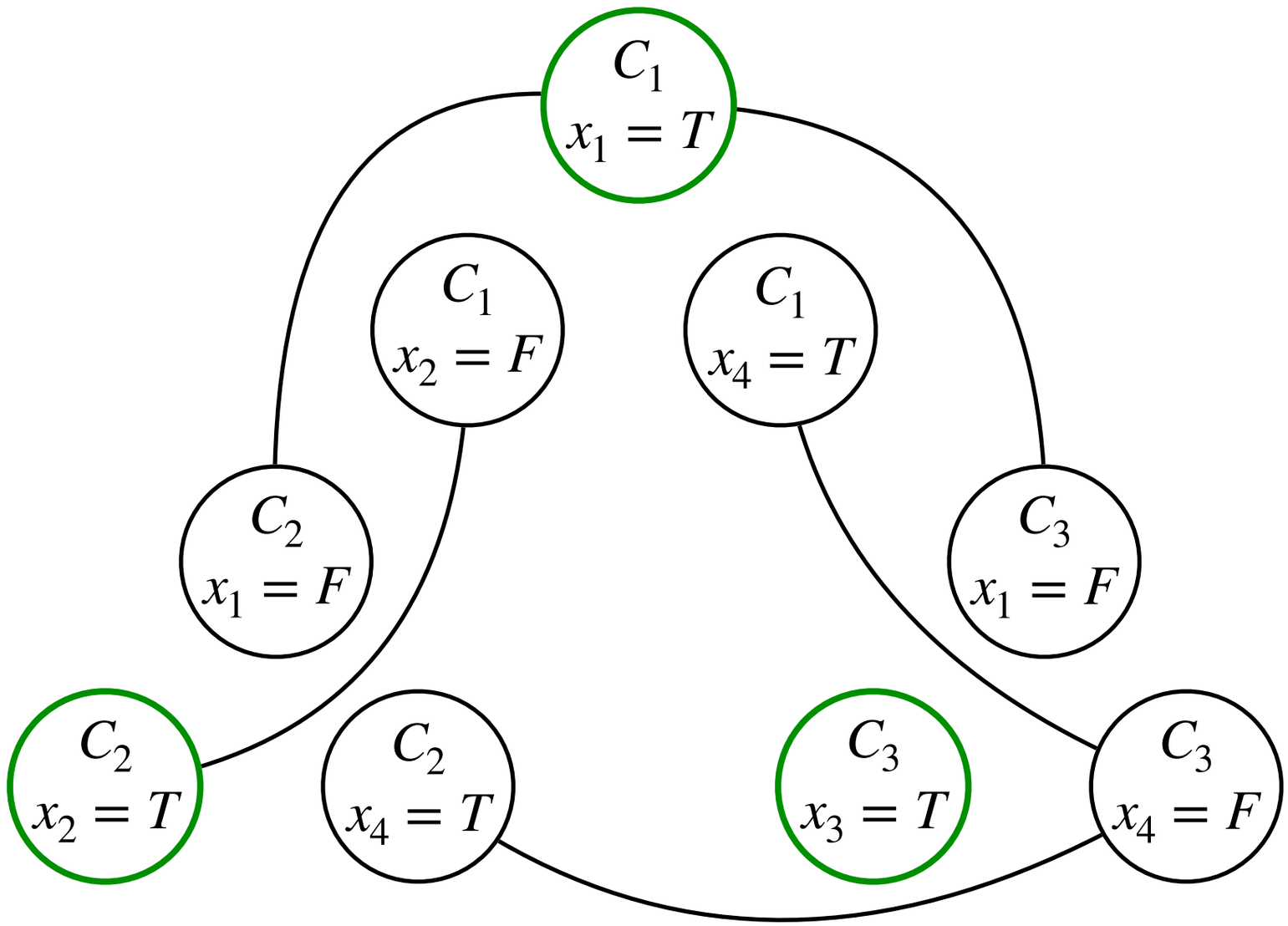
let m be the number of clauses

$$\underbrace{(x_1 \vee \bar{x}_2 \vee x_4)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}_{C_3}$$

Now, suppose there is a satisfying assignment.

Then in each clause, at least one vertex's variable setting is consistent with the assignment.

Selecting one vertex per clause, we have size-m subset of vertices.

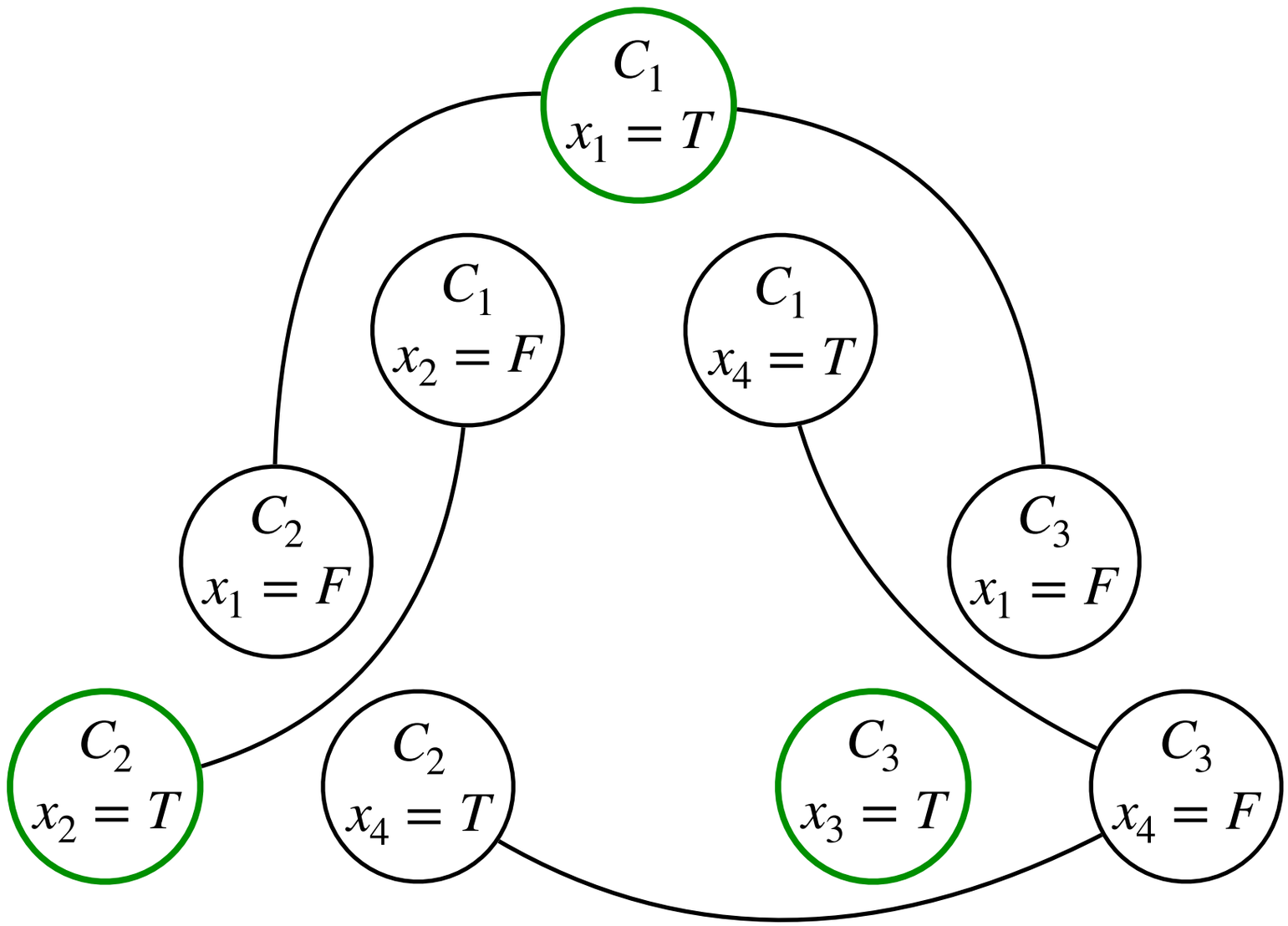


3-SAT Reduces to Independent Set

let m be the number of clauses

$$\underbrace{(x_1 \vee \bar{x}_2 \vee x_4)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}_{C_3}$$

Moreover, this subset is an independent set since edges represent conflicting variable settings.



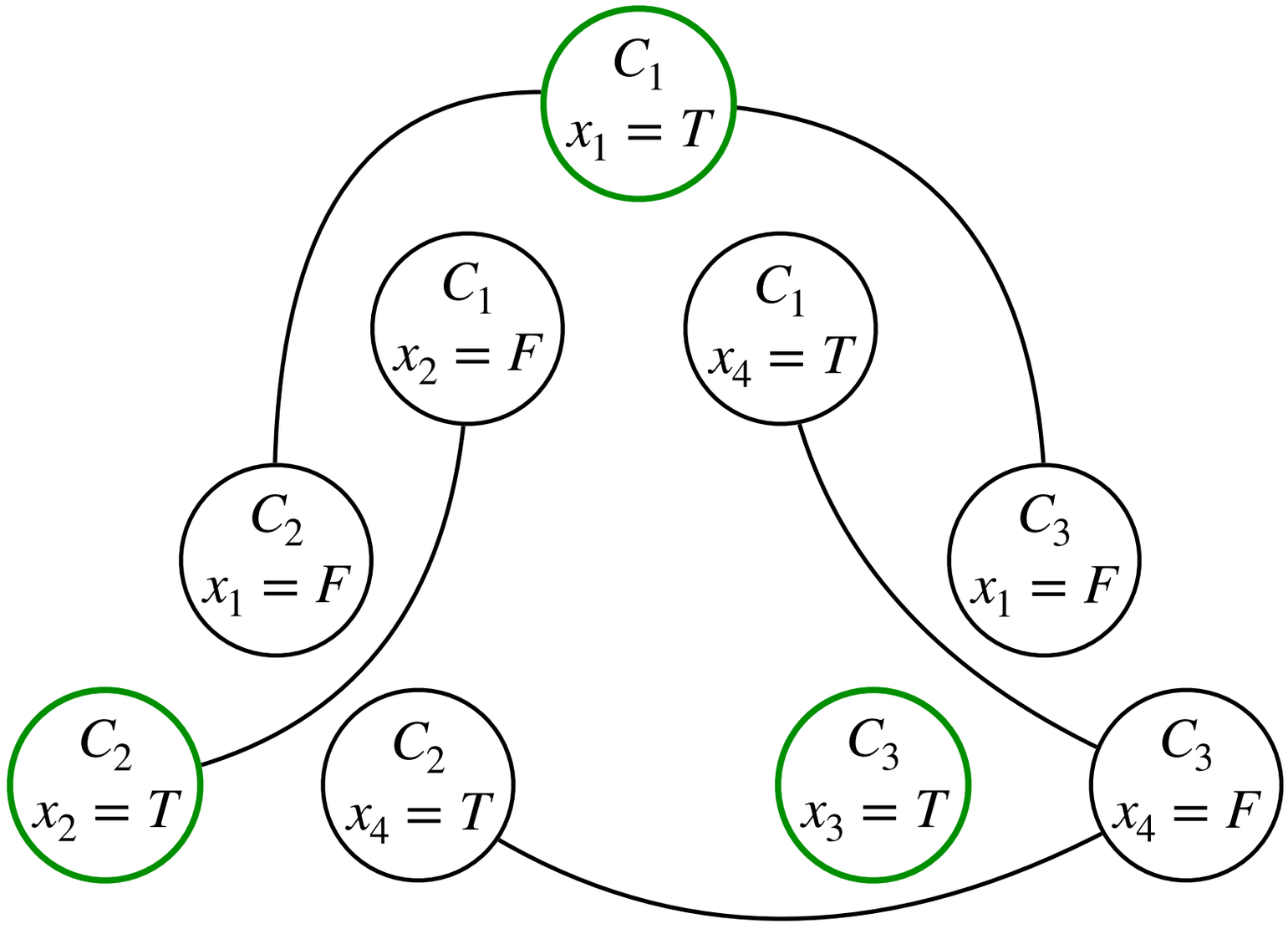
3-SAT Reduces to Independent Set

let m be the number of clauses

$$\underbrace{(x_1 \vee \bar{x}_2 \vee x_4)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}_{C_3}$$

Idea: If there is an independent set of size at least m, return as a satisfying assignment any assignment consistent with that independent set.

So far, we have shown that if there is a satisfying assignment, then our reduction correctly finds such an assignment.

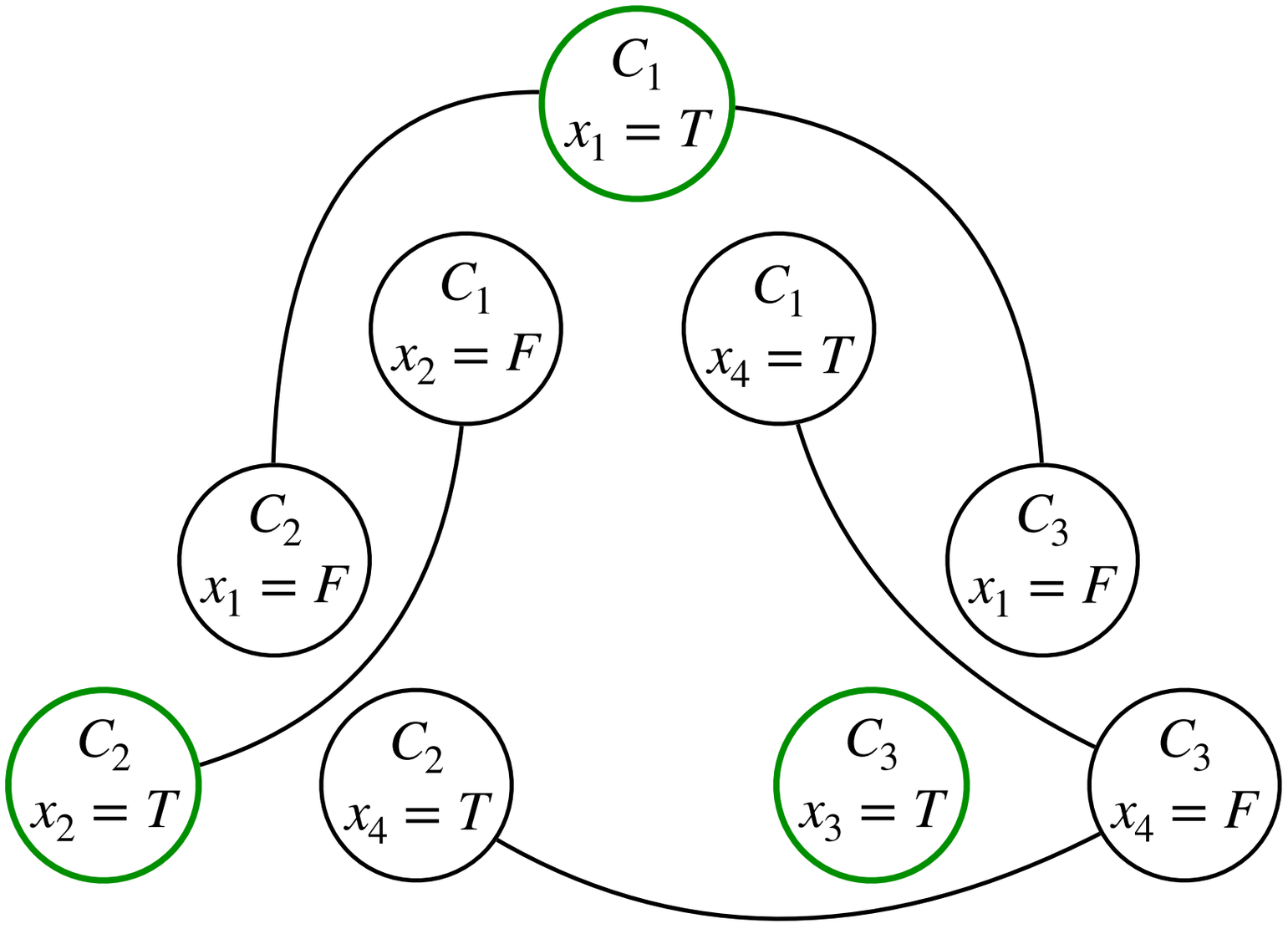


3-SAT Reduces to Independent Set

let m be the number of clauses

$$\underbrace{(x_1 \vee \bar{x}_2 \vee x_4)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}_{C_3}$$

But we also need to show that if there is not a satisfying assignment, then our reduction correctly determines this (i.e., no independent set contains at least m vertices).



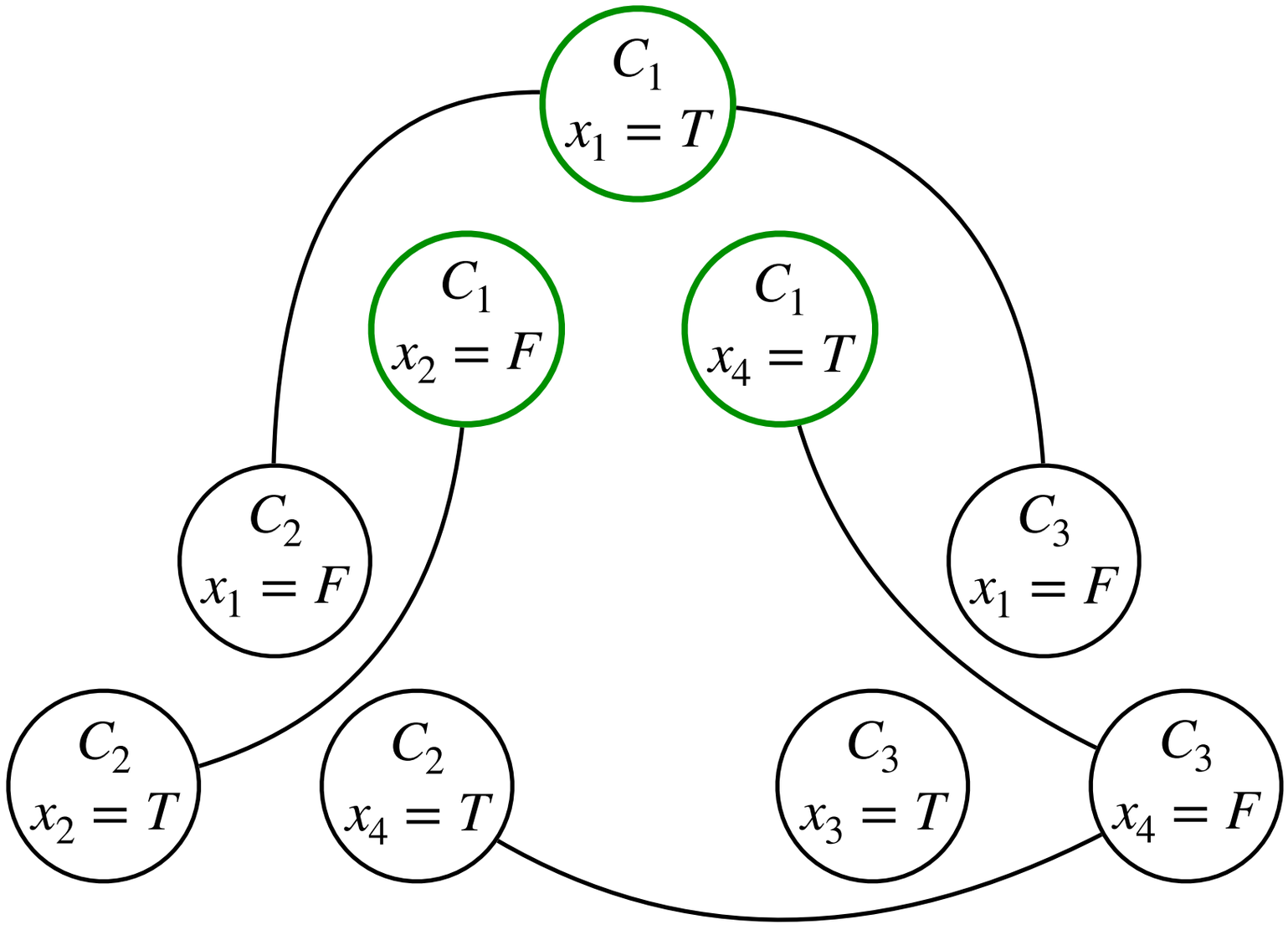
3-SAT Reduces to Independent Set

let m be the number of clauses

$$\underbrace{(x_1 \vee \bar{x}_2 \vee x_4)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}_{C_3}$$

But we also need to show that if there is not a satisfying assignment, then our reduction correctly determines this (i.e., no independent set contains at least m vertices).

Uh-oh!



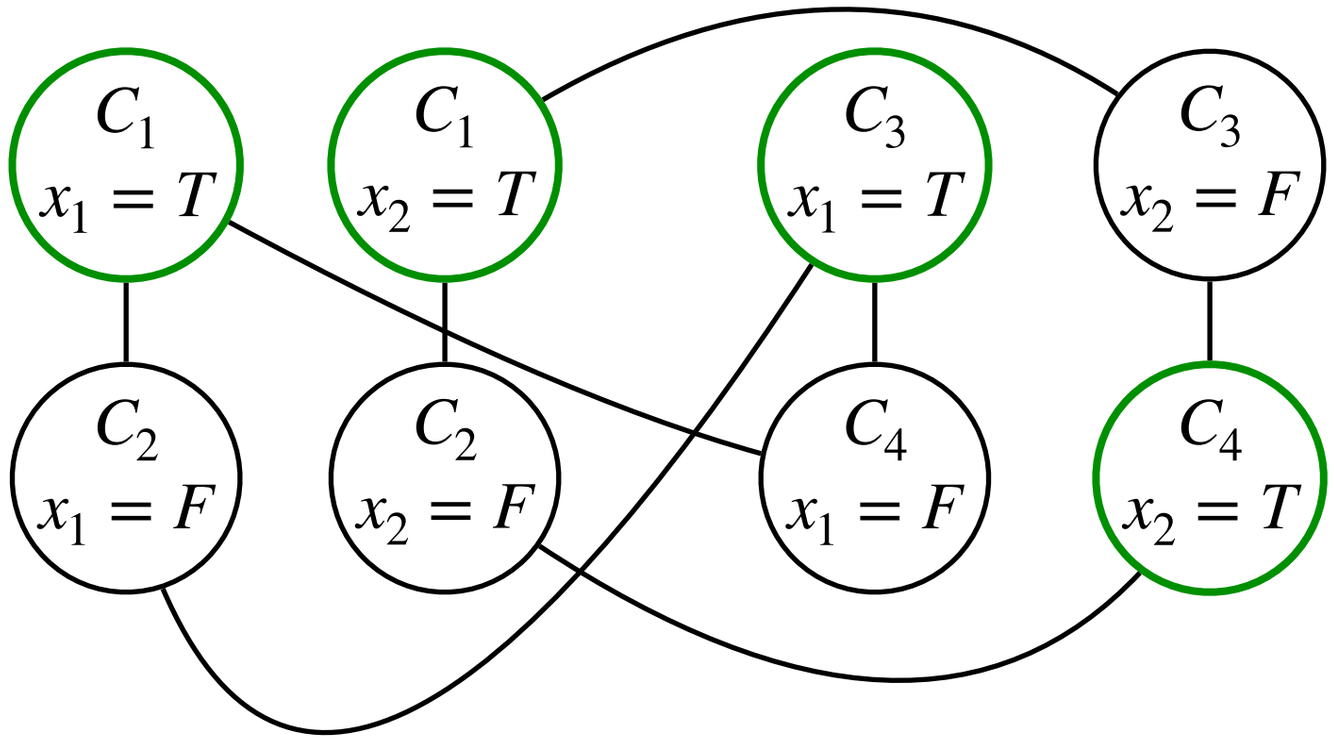
3-SAT Reduces to Independent Set

let m be the number of clauses

$$\underset{C_1}{(x_1 \vee x_2)} \wedge \underset{C_2}{(\bar{x}_1 \vee \bar{x}_2)} \wedge \underset{C_3}{(x_1 \vee \bar{x}_2)} \wedge \underset{C_4}{(\bar{x}_1 \vee x_2)}$$

This problem is a real problem. Consider a different, unsatisfiable instance. There is an independent set of size $m = 4$ despite no assignment being feasible.

The issue is that we haven't forbidden including more than one vertex per clause.



3-SAT Reduces to Independent Set

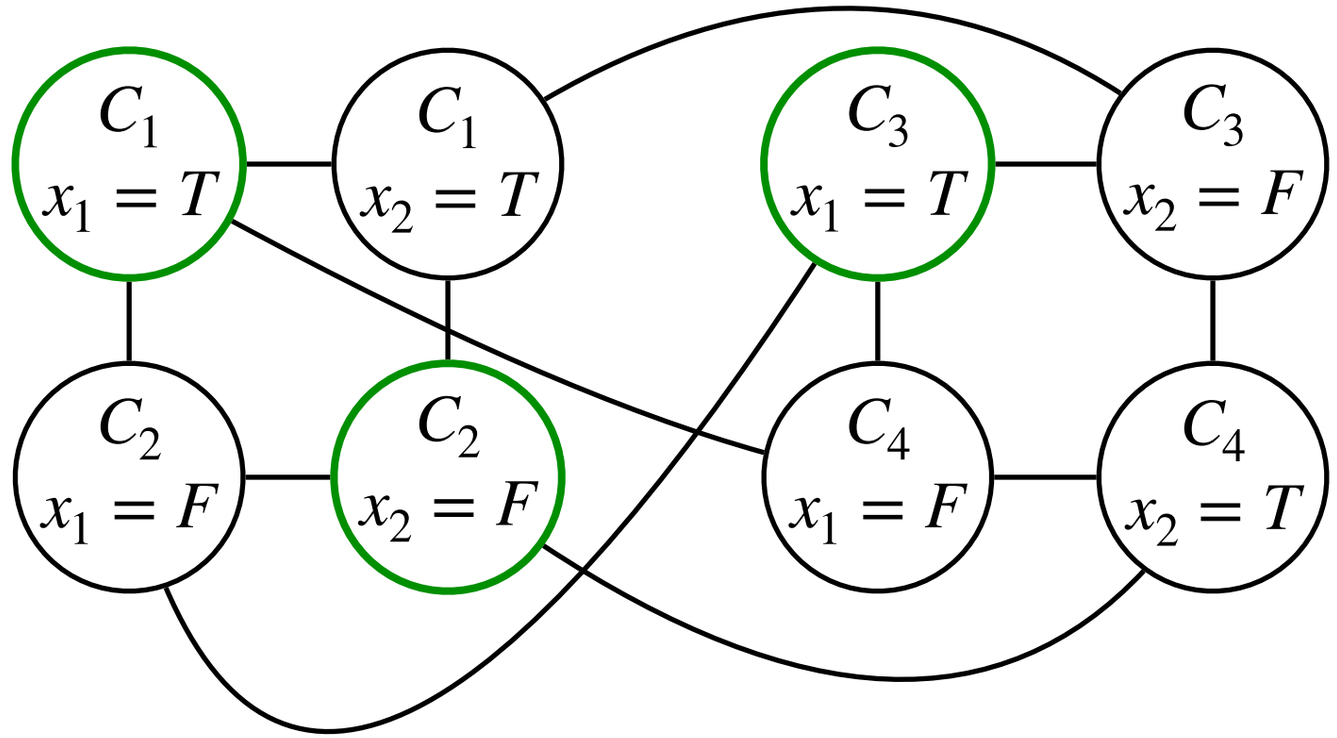
let m be the number of clauses

$$\underset{C_1}{(x_1 \vee x_2)} \wedge \underset{C_2}{(\bar{x}_1 \vee \bar{x}_2)} \wedge \underset{C_3}{(x_1 \vee \bar{x}_2)} \wedge \underset{C_4}{(\bar{x}_1 \vee x_2)}$$

The issue is that we haven't forbidden including more than one vertex per clause.

Solution: Make each clause's vertices a clique.

Now no independent set is larger than
$3 = m - 1$

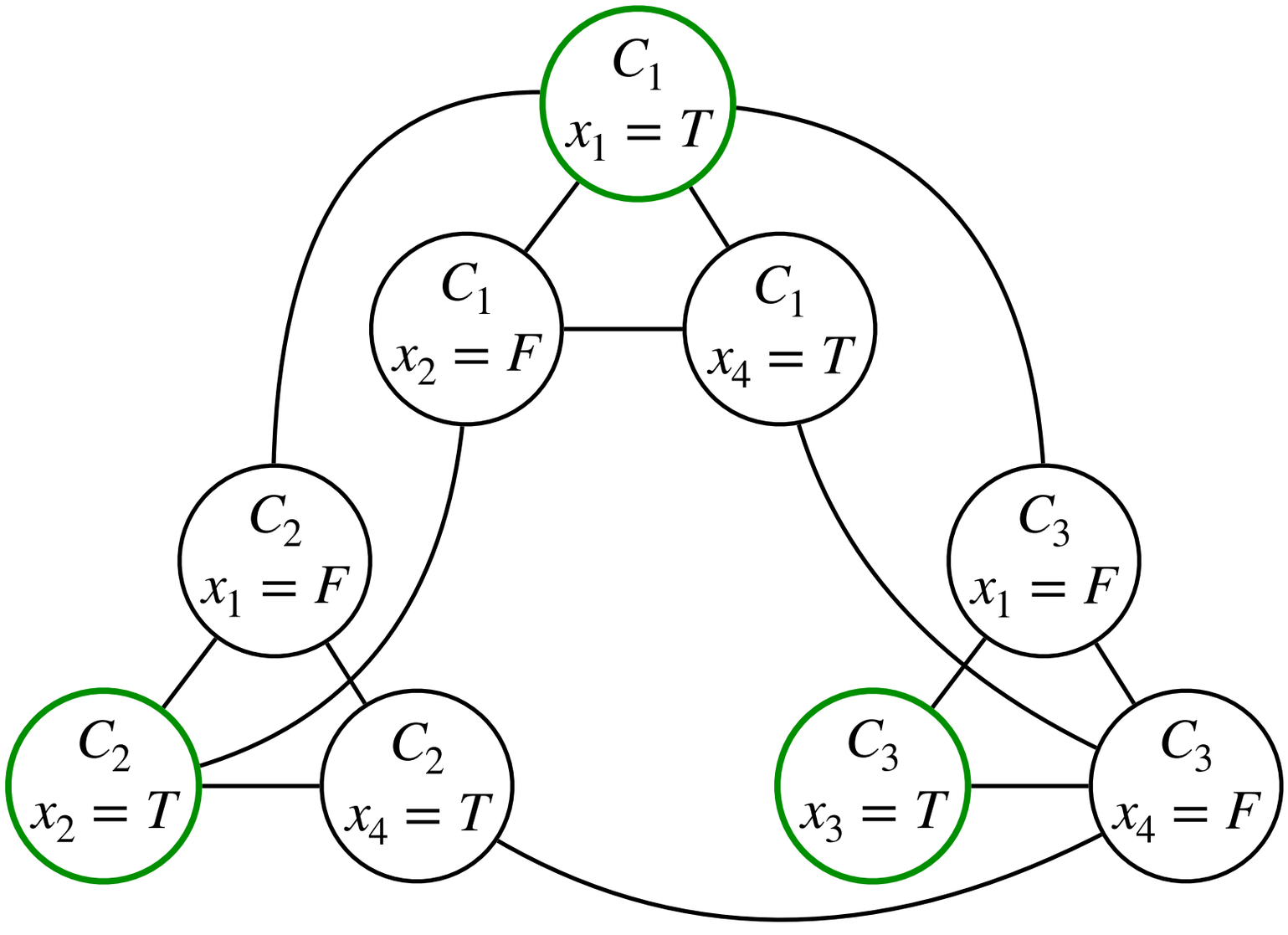


3-SAT Reduces to Independent Set

let m be the number of clauses

$$\underset{C_1}{(x_1 \vee \bar{x}_2 \vee x_4)} \wedge \underset{C_2}{(\bar{x}_1 \vee x_2 \vee x_4)} \wedge \underset{C_3}{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}$$

Coming back to the original example, with each clause's vertices being a clique, we have the updated diagram on the right



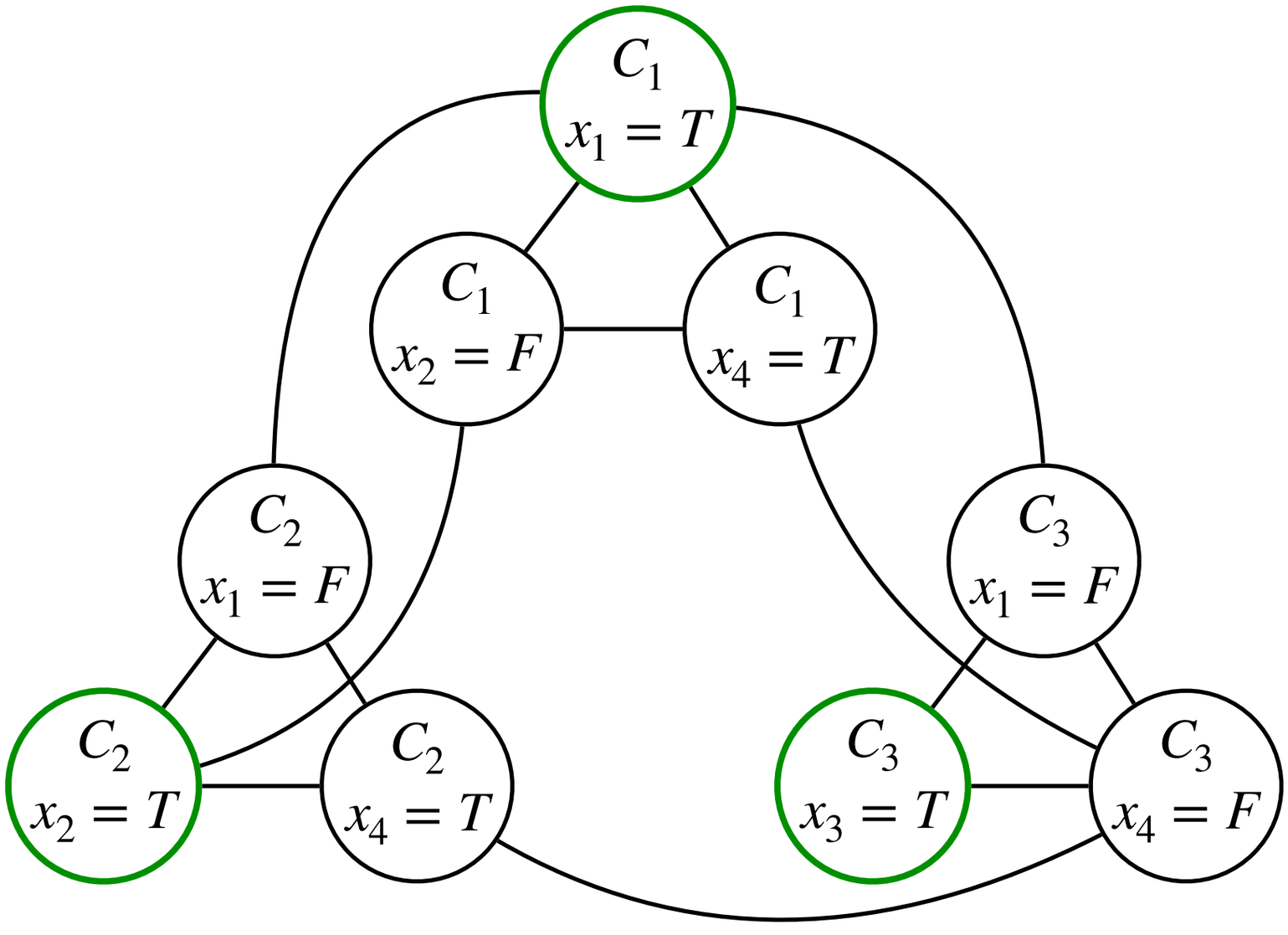
3-SAT Reduces to Independent Set

let m be the number of clauses

$$\underbrace{(x_1 \vee \bar{x}_2 \vee x_4)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}_{C_3}$$

(Recall)
Idea: If there is an independent set of size at least m, return as a satisfying assignment any assignment consistent with that independent set.

It still holds that if there is a satisfying assignment, then our reduction correctly finds such an assignment.

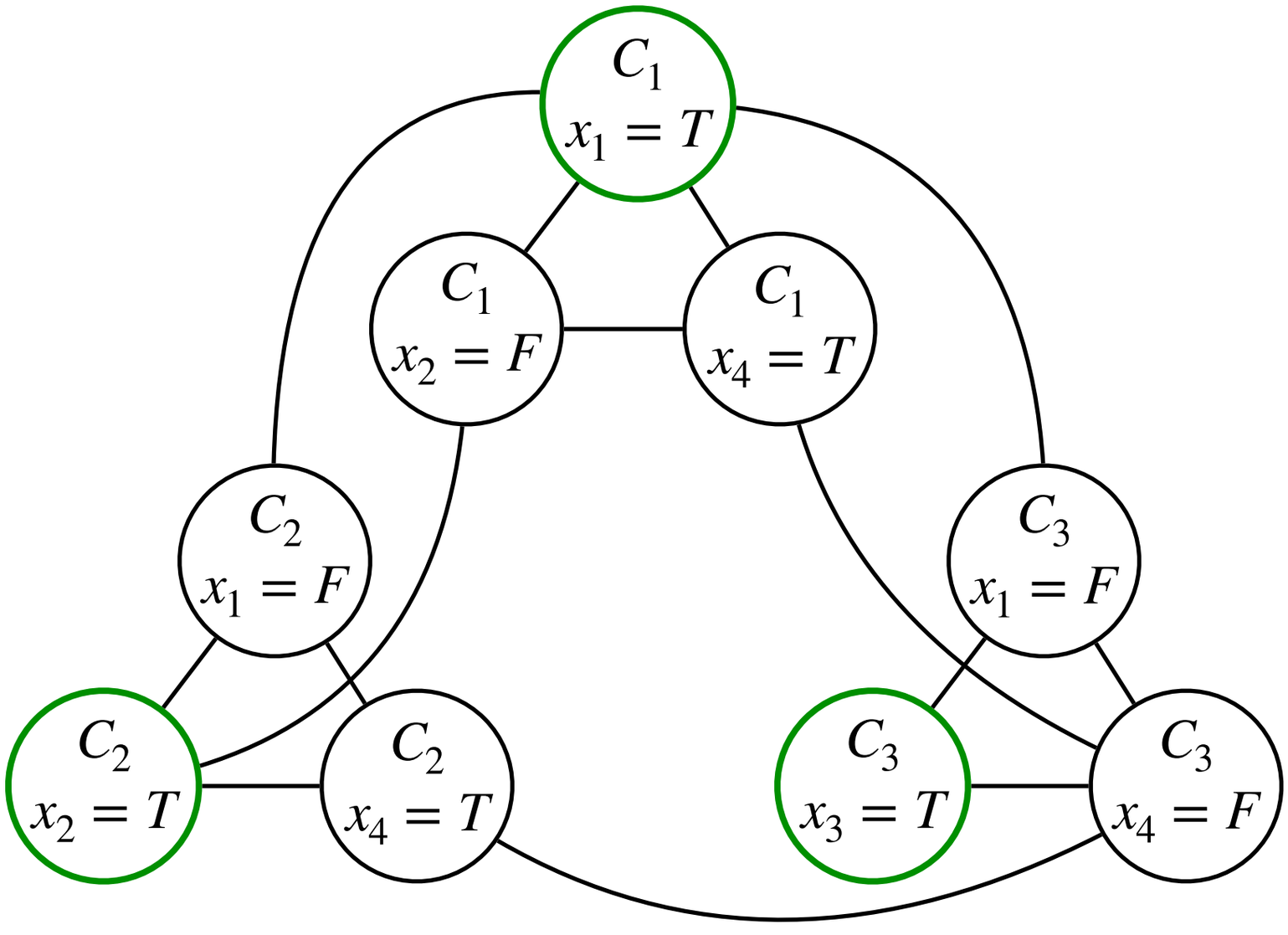


3-SAT Reduces to Independent Set

let m be the number of clauses

$$\underbrace{(x_1 \vee \bar{x}_2 \vee x_4)}_{C_1} \wedge \underbrace{(\bar{x}_1 \vee x_2 \vee x_4)}_{C_2} \wedge \underbrace{(\bar{x}_1 \vee x_3 \vee \bar{x}_4)}_{C_3}$$

Suppose there is no satisfying assignment, yet we find an independent set of size at least m. There can be at most one vertex per clause, so the independent set must be of size m and contain one vertex per clause. But there is an assignment consistent with the independent set which satisfies all clauses, a contradiction!



3-SAT Reduces to Independent Set

We have given the following Levin reduction (which we proved is correct):

- Does the following preprocessing (forming a graph):
  - Create one vertex per clause per literal
  - Add an edge between conflicting literals
  - Add an edge between all vertices from the same clause
- Calls subprocedure for Search version of Independent Set (specifically, search for an independent set of size at least m)
- Does the following postprocessing:
  - If the subprocedure finds a feasible solution (an independent set of size m), return any assignment consistent with the independent set.
  - If the subprocedure returns "no solution", the return "no solution" (indicating the formula is not satisfiable)

NP-Hardness is a bit coarse

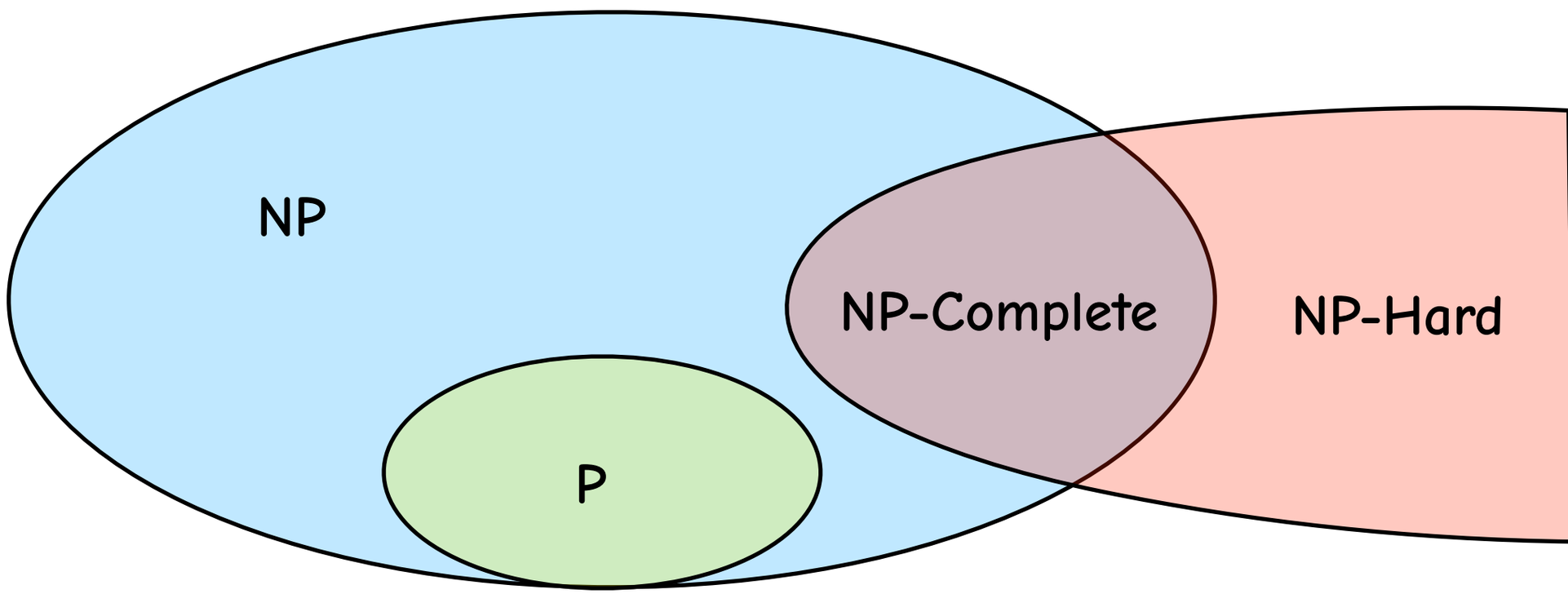
- NP-Hardness is a useful descriptor to indicate that a problem is likely hard
- However, some NP-Hard problems could be much more difficult than others:
  - It is possible (to our knowledge) that there exist NP-Hard problems which require exponential time
  - Yet, there are NP-Hard problems that we know can be solved in subexponential time solutions

- Can we restrict to a smaller class of problems that are all essentially equivalent (by equivalent, we mean "can be reduced to each other in polynomial time")? Yes!

NP-Completeness

- Problem B is NP-Complete if both:
  - B is in $\mathbf{NP}$
  - For every problem A in $\mathbf{NP}$, there is a Levin reduction from A to B

Venn diagram
if $\mathbf{P} \neq \mathbf{NP}$

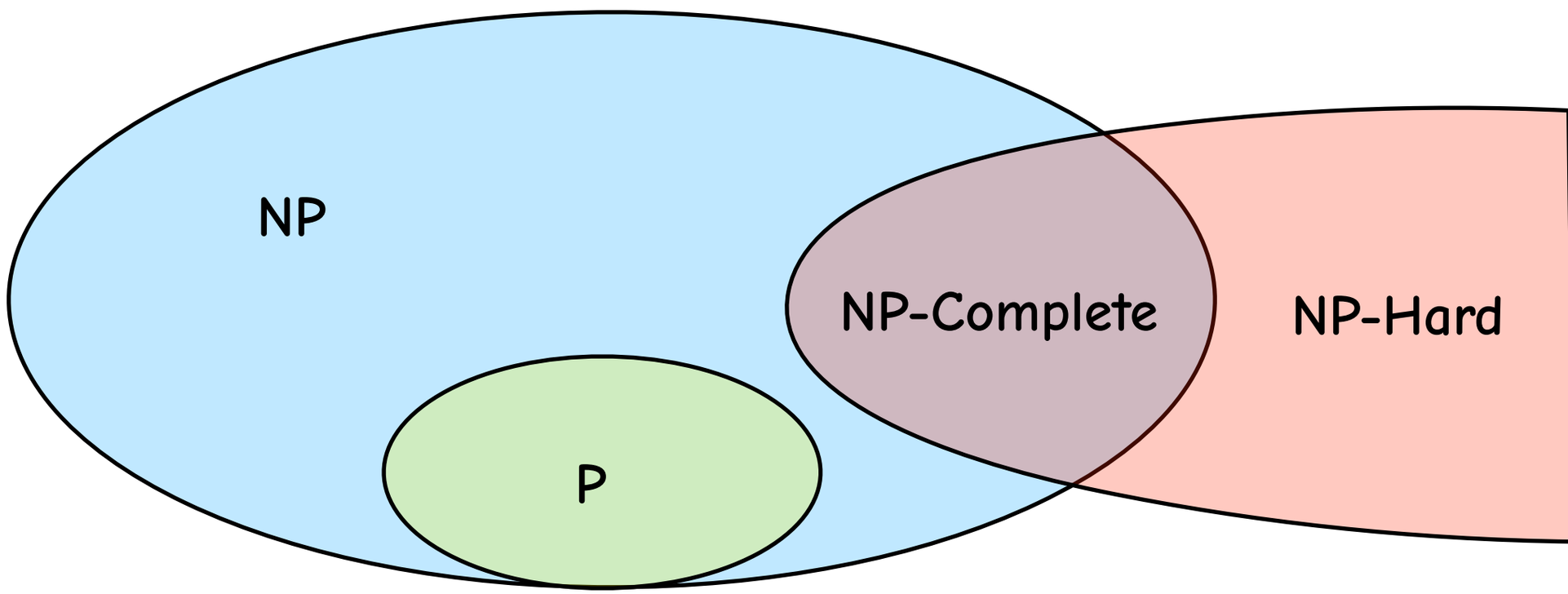


NP-Completeness

- Problem B is NP-Complete if both:
  - B is in $\mathbf{NP}$
  - For every problem A in $\mathbf{NP}$, there is a Levin reduction from A to B

- NP-complete problems are the hardest problems in $\mathbf{NP}$

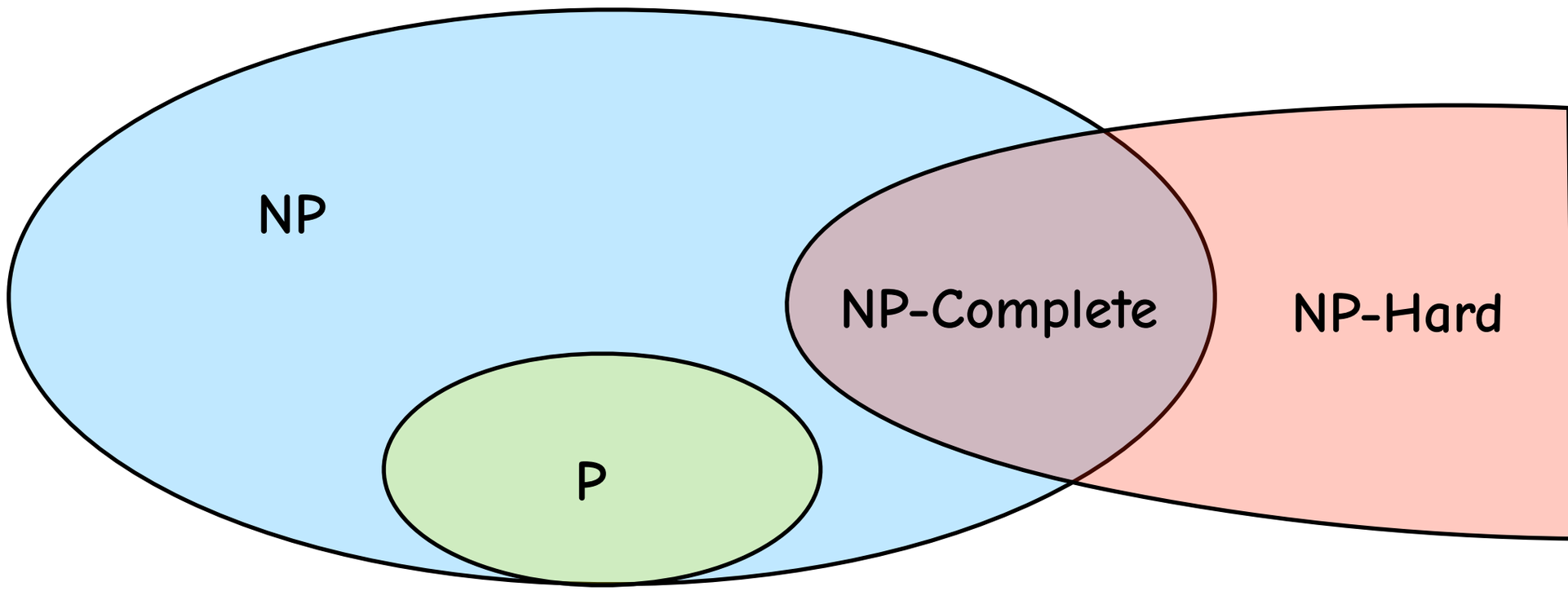
Venn diagram if $\mathbf{P \neq NP}$



NP-Completeness

- Problem B is NP-Complete if both:
  - B is in $\mathbf{NP}$
  - For every problem A in $\mathbf{NP}$, there is a Levin reduction from A to B
  - Exercise: Verify that if A and B are NP-complete, then A reduces to B. So, all NP-complete problems are essentially equivalent!

Venn diagram if $\mathbf{P} \neq \mathbf{NP}$



How to prove that a problem is NP-Complete

- How can we prove that a problem B is NP-Complete?
- 3-step recipe:

  1) Prove that B is in $\mathbf{NP}$

  2) Identify an NP-Complete problem A

  3) Use a Levin reduction to reduce A to B

NP-Hard versus NP-Complete

- All the NP-Hard problems we considered are in $\mathbf{NP}$, including 3-SAT

- Any problem in NP reduces to 3-SAT via a Levin reduction

    - This is what the proof of the Cook-Levin Theorem actually shows

- So, 3-SAT is NP-Complete

- It turns out that all the previous NP-Hard problems we considered reduce to 3-SAT via a Levin reduction, so these problems are all NP-Complete!

The Practical Side

- A lot of effort has been put into developing SAT solvers
- SAT solvers:
  - often are able to provide solutions to SAT problems
  - but can sometimes fail to do so (after all, SAT is NP-Hard)
- So what? Well, technically, any NP-Complete problem can be reduced to 3-SAT. If you can (without too much pain / loss of efficiency) encode your problem as a 3-SAT problem, then you can try seeing what a SAT solver can do for your problem
- Links:
  - International SAT competition
  - MiniSAT

Deeper Theoretical Questions

- If $\mathbf{P} \neq \mathbf{NP}$, must it be the case that NP-Hard problems require exponential time? We don't know

- **Exponential Time Hypothesis (ETH):**

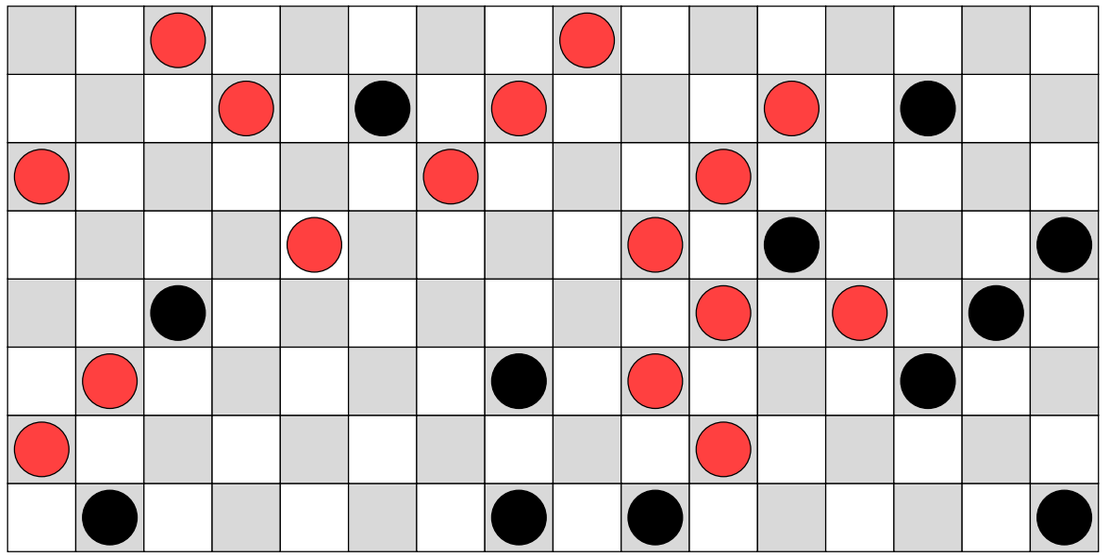
  - There is a constant $c > 1$ such that for any algorithm that solves 3-SAT, at least c^n time is required.

- Note that ETH is a conjecture. It has not been proven!

Deeper Theoretical Questions

- Are there any problems for which we know exponential time is required?

- Yes! Checkers (generalized to an $m \times n$ board)



- Checkers Decision problem: Given a particular board state, determine whether it is possible for red to force a win

- This problem requires time exponential in m and n

Back to P ≠ NP conjecture

- Recall the conjecture that $\mathbf{P} \neq \mathbf{NP}$
- We still have no idea whether the conjecture is true
- What would it mean if the conjecture is false (so $\mathbf{P} = \mathbf{NP}$)?
  - Then verifying a solution is as easy as finding a solution
    - But this would be quite surprising because verifying seems to require no creativity, while finding a solution seems to require a lot of creativity
- If $\mathbf{P} \neq \mathbf{NP}$ as most experts suspect, why hasn't anyone managed to prove it yet. Potential reasons:
  - In general, we are not as good at proving lower bounds (limitations of algorithms) compared to upper bounds
  - Maybe it's just hard to prove it!